Report

Subject : Tecnologie e Applicazioni Web

Author : Pagano Matteo 880833

Report Structure

1 Introduction

For the “Tecnologie e Applicazioni Web” 2022/2023 exam, the goal is to develop a full-stack application that offers a comprehensive solution for restaurant management.

In this report I will provide an overview of the application’s architecture and demonstrate how I implemented all the specified requirements.

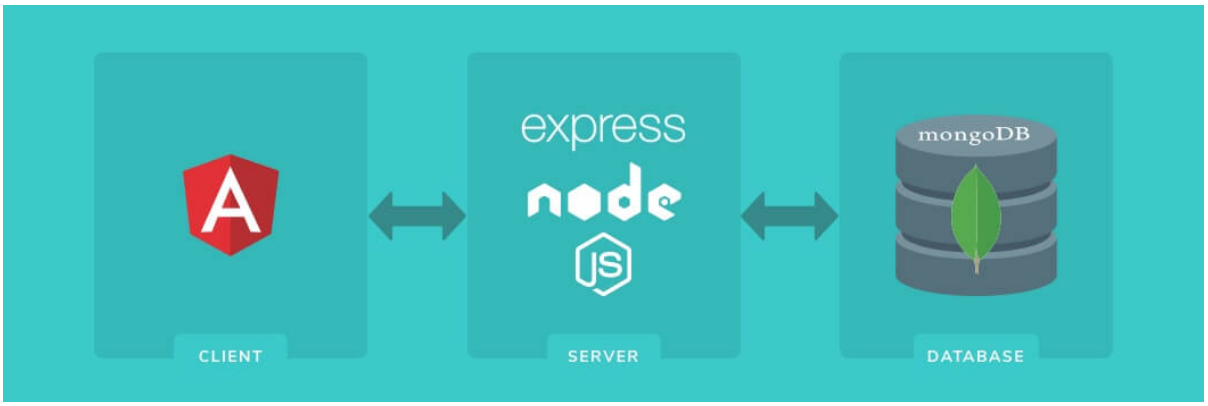
In conclusion I would like to spend time to give some personal opinions regarding the solutions and what could be the future developments.

For storing the project I used the versioning control system GIT. This is the link to the GitHub’s page of the project’s folder.

1.1 System architecture

In this section I want to introduce to the reader how the entire architecture of the application is structured.

For having a first general compension of application’s architecture I show below a simple image of how It’s composed.



How we can see there are three main component, a database for data persistence, a backend server for the application’s logic and a client-frontend for the data visualization.

In particular for the development of the Frontend component It’s been used the Angular Framework, and for the Backend It’s been used the Express framework inside a node.js development’s environment.

This two framework are commonly used together due to the fact that both are written with the JavaScript language, and so the application can be written using only one programming language.
In particular, the JavaScript code is the final product of the TypeScript object oriented language (that is a supertype of the JavaScript language) when It’s compiled.

1.1.1 Component communication

How does the communication between these components take place?
For answer this question I have to introduce the HTTP protocol. The HTTP protocol is a network protocol used for the communication between calculators throught the net.
How the HTTP protocol works?
The communication of the HTTP protocol happens in two steps:

1) The sender sents a request to the receiver specifying the url ,

2) The receiver sents a response to the sender with the information requested.

This communication is unidirectional, infact the connection it's established one time and then closed.
In the Tecnologie e Applicazioni Web project, as we can see in the image above, this kind of communication happens between the Frontend and the Backend, and between the Backend and the Database. There is no direct communication between the Frontend and the Database itself.

Real Time Communcation

For the real-time communication between the clients of the Angular component, it's been used the Web Socket protocol. The WebSocket protocol is a bidirectional connection, infact, it's establish only the first time, and is persistent until the client decides to stop it.
In the Tecnologie e Applicazioni Web project, this protocol is used for the real-time communication between client, like the notify of the orders that are been completed.

Both two kind of communication, in Angular and in Express components, use the data form JSON.

1.1.2 Docker for the deployment

For the deployment of the applications, I decide to use Docker and Docker-Compose for the build and the boot of each components, making easier the deployment and the boot of the entire application.

Belove, the code for the bootstrap and the build of the image associated with each component.

Frontend Dockerfile

```
FROM node:16 as build
WORKDIR /app/Frontend
COPY package*.json ./
RUN npm install
COPY . .
RUN npm run build
FROM nginx:alpine
COPY --from=build /app/Frontend/dist/restaurant /usr/share/nginx/html
COPY my-angular-app.conf /etc/nginx/conf.d/
RUN rm /etc/nginx/conf.d/default.conf
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

Backend Dockerfile

```
FROM node:14
RUN mkdir -p /app/Backend
WORKDIR /app/Backend
COPY package.json package-lock.json /app/Backend/
RUN npm install
COPY /compiledSourceJS/ /app/Backend/compiledSourceJS
COPY /tsconfig.json /app/Backend/
EXPOSE 3000
CMD ["node", "./compiledSourceJS/Backend/server.js"]
```

Docker-compose

```
version: '3.8'
services:
mongodb:
image: mongo:latest
container_name: mongodb
volumes:
- MongoVolume:/data/db
```

```
networks:
- taw-subnet

backend:
build: ./Backend
container_name: backend
ports:
- "3000:3000"
depends_on:
- mongodb
#volumes:
# - ./Backend:/app/Backend
networks:
- taw-subnet

frontend:
build: ./Frontend/restaurant
container_name: frontend
ports:
- "80:80"
depends_on:
- mongodb
- backend
networks:
- taw-subnet

networks:
taw-subnet:
external: true

volumes:
MongoVolume:
```

How we can see each component is linked to a net called “taw-subnet”, and can now communicate with each component.

2 Component Architecture

In this section i will show in a detailed way how each components are composed and the main aspects that in which I’ve been focalized.

2.1 Backend

The Backend component is the main logic of the entire system. It is collocated between the Frontend and the Database components, infact It receive request from the Frontend component and retrieve data from the Database.

2.1.1 Express

The framework used for the developing of the backend component is Express. The Express’s structure of the project is divided in three parts, the middlewares, the endpoints, and the server. I decided to structure each routes concatening many middlewares, in order to reach the final endpoint function.

In fact the main structure of each routes, is :

1) routes URL
2) One or more middlewares
3) Final Ednpoint

There is a little example of what I mean

```
app.get('/restaurants/:idr/tables/:idt/group/orders', MW.verifyJWT,
MW.isWorkerOfThisRestaurant, MW.isTableOfThatRestaurant, MW.tableHasAGroup,
EP.getOrdersByRestaurantAndTable);
```

How we can see the API that I expose in the server is the retrieve of all orders of a particular group sitting in a particular table of a restaurant. For reaching the final function getOrdersByRestaurantAndTable, that it will have the purpose of retrieve from the database the data, the request of the Angular client must go through multiplies controlles, like the verify of isWorkerOfThisRestaurant and isTableOfThatRestaurant and so on.

This kind of implementation of the express server's routes makes easier for me to develop any single endpoint, thanks to the fact that the various MW.* functions can be reused in many others routes, an is easier to understand where to put new code for modifying the application.

2.1.2 Common Middlewares

This section show to the reader which are the common middleware used in each request comes.

2.1.2.1 Body Parser middleware

All the requests that come throught the HTTP protocol to the express server, are parsed with the bodyParser.json() function. It has the task of parse the body content's request in JSON format and inject it in the body property of the req parameter.

This middleware allow me the ability of access the properties of the JSON request in a object oriented way. this is a example of how it works:

JSON Body of the request:

```
{
  "items" : [
     {"itemId" : "64d6a5e10884c1758a94dae9", "count" : "2"},
     {"itemId" : "64d6a5ee0884c1758a94daf3", "count" : "3"},
     {"itemId" : "64d6a5f60884c1758a94daf9", "count" : "1"}
  ]
}
```

Req.body property:

```
const itemsList: ItemRequest = req.body;

for (const itemData of itemsList.items) {
    const itemId = itemData.itemId;
    console.log(itemId)
```

Terminal

```
64d6a5e10884c1758a94dae9
64d6a5ee0884c1758a94daf3
64d6a5f60884c1758a94daf9
```

2.1.2.2 Log middleware

Each request meets the “log” middleware that has the task of printing the method and the url associated with it

```
app.use((req, res, next) => {
  console.log(`Nuova richiesta in entrata: ${req.method} ${req.url}`);
  next();
});
```

Terminal

```
Nuova richiesta in entrata: GET /restaurants/64e9f53c2fc327c9b3c97977/items
```

2.1.2.3 Cors middleware

When a request comes from a client, the server response with determinants intestations.

```
app.use(cors());
```

In this case, the intestations attached are access-Control-Origin : * which indicate that all origin can access to the server resources. There are also many others intestations added.

2.1.2.4 Errors Management

If any route corrispond to the request coming, there is a special endpoint that is invoked:

```
app.use( (req,res,next) => {
  res.status(404).json({statusCode:404, error:true, errormessage: "Invalid endpoint"} );
})
```

else, if any middleware or endpoint catch an error, this endpoint is invoked:

```
app.use( function(err : any, req : Request, res : Response, next : NextFunction) {
  console.log("Request error: " + JSON.stringify(err) );
  res.status( err.statusCode || 500 ).json( err );
});
```

2.1.3 APIs exposes

For clarifying which are the Routes exposes from the Express Server, I list below all the APIs recallable from outside the Express context, with a short explaination of what they do, and what are the parameters and little examples of data exchanged.
Before the string of the Route, there is also the `"/api/v1"` string to add.

2.1.3.1 Users management (Only Owner) APIs

HTTP Method	Route	Description
GET	`/restaurants/:idr/cooks`	Retrieve a list of cooks at the restaurant

GET	/restaurants/:idr/waiters	Retrieve a list of waiters at the restaurant
GET	/restaurants/:idr/cashiers	Retrieve a list of cashiers at the restaurant
GET	/restaurants/:idr/bartenders	Retrieve a list of bartenders at the restaurant
POST	/restaurants/:idr/cooks	Add a new cook to the restaurant
POST	/restaurants/:idr/waiters	Add a new waiter to the restaurant
POST	/restaurants/:idr/cashiers	Add a new cashier to the restaurant
POST	/restaurants/:idr/bartenders	Add a new bartender to the restaurant
DELETE	/restaurants/:idr/cooks/:idu	Remove a specific cook from the restaurant
DELETE	/restaurants/:idr/waiters/:idu	Remove a specific waiter from the restaurant
DELETE	/restaurants/:idr/cashiers/:idu	Remove a specific cashier from the restaurant
DELETE	/restaurants/:idr/cashiers/:idu	Remove a specific bartender from the restaurant

For simplicity, I write only the code about one kind of user, infact all GET, POST and DELETE methods are similar for each.

GET Response


```
{
"error": false,"errormessage": "",
"cooks": [
{
"_id": "64e9f8f72fc327c9b3c97be2",
"itemsPrepared": [{"idItem": "64e9f6b82fc327c9b3c97a44","count": 5}],
"username": "Graziano",
"email": "graziano@gmail.com",
"role": "cook",
"idRestaurant": "64e9f53c2fc327c9b3c97977",
"salt": "84a6195f3508fb77722fdf42613050da",
"digest":
"290a035e4d9eefb0f045ba8b137ed3247779dea4dfdcedc6cc6235f7d42721b790b27397c7036ee1965e3fe2be5869316e9
fbbaca0b3e1b7dfed2f2c399c8dfc",
"__v": 14
}
]
}
```


POST, Body Example

```
{
  "username" : "cook1",
  "email" : "email1@gmail.com"
}
```

Response


```
{
"error": false,
"errormessage": "",
"idNewCook": "64da4f75f6d6b0438e0ff592",
"usernameNewCook": "cook1",
"email": "email1@gmail.com",
"passwordToChange": "Jh4Cpjzl"
}
```

```
DELETE Response
{
  "error": false,
  "errormessage": "",
  "idCookDeleted": "64e9f9122fc327c9b3c97c02"
}
```

2.1.3.2 Tables APIs

HTTP Method	Route	Query	Description
GET	/restaurants/:idr/tables	isFull : true or false	Retrieve a list of tables for the restaurant

```
Example Response

{
  "error": false,
  "errormessage": "",
  "tables": [
    {
      "_id": "64e9f9e12fc327c9b3c97ca6",
      "tableNumber": "Tavolo 1",
      "maxSeats": 10,
      "group": null,
      "restaurantId": "64e9f53c2fc327c9b3c97977",
      "__v": 0
    }
  ]
}
```

POST	/restaurants/:idr/tables		Add a new table to the restaurant

Body example

```
 {
   "tableNumber" : "Tavolo 2",
   "maxSeats" : "20"
 }
```

Response

```
{
    "error": false,
    "errormessage": "",
    "table": {
        "tableNumber": "Tavolo 2",
        "maxSeats": 20,
        "group": null,
        "restaurantId": "64e9f53c2fc327c9b3c97977",
        "_id": "64f821a3304ffe45e64ec806",
        "__v": 0
    }
}
```

DELETE	/restaurants/:idr/tables/:idt		Remove a specific table from the restaurant

```
Example Response
{
  "error": false,
  "errormessage": "",
  "idTableDeleted": "64f821a3304ffe45e64ec806"
}
```

2.1.3.3 Items APIs

HTTP Method	Route	Query	Description
GET	/restaurants/:idr/items	type : dish or drink	Retrieve a list of items for the restaurant

Response


```
{
  "error": false,
  "errormessage": "",
  "items": [
  {
    "_id": "64e9f61d2fc327c9b3c979ae",
    "itemName": "Crispy Truffle Salmon",
    "itemType": "dish",
    "price": 8,
    "preparationTime": 5,
    "idRestaurant": "64e9f53c2fc327c9b3c97977",
    "countServered": 80,
    "__v": 0
    }
  ]
}
```


POST	/restaurants/:idr/items		Add a new item to the restaurant

Body example


```
{
  "itemName" : "Carbonara",
  "itemType" : "dish",
  "price" : "22",
  "preparationTime" : "5"
}
```


Response


```
{
    "itemName": "Carbonara",
    "itemType": "dish",
    "price": 22,
    "preparationTime": 5,
    "idRestaurant": "64e9f53c2fc327c9b3c97977",
    "countServered": 0,
    "_id": "64f82316304ffe45e64ec8b9",
    "__v": 0
}
```


DELETE	/restaurants/:idr/items		Remove a specific item from the restaurant

Response


```
{
  "error": false,
  "errormessage": "",
  "idItemDeleted": "64f82316304ffe45e64ec8b9"
}
```

2.1.3.4 Customer Groups APIs

<table>
<tr><th>HTTP Method</th><th>Route</th><th>Description</th></tr>
<tr><td>GET</td><td>/restaurants/:idr/tables/:idt/group</td><td>Retrieve the customer group associated with a table</td></tr>
<tr><td colspan="3">Response
{
"error": false,
"errormessage": "",
"group": {
"_id": "64f82781304ffe45e64ec8fd",
"numberOfPerson": 1,
"dateStart": "2023-09-06T07:17:21.470Z",
"dateFinish": null,
"ordersList": [],
"idRestaurant": "64e9f53c2fc327c9b3c97977",
"idRecipe": null,
"idTable": "64e9f9e12fc327c9b3c97ca6",
"__v": 0
}
}</td></tr>
<tr><td>POST</td><td>/restaurants/:idr/tables/:idt/group</td><td>Add a new customer group to a table</td></tr>
<tr><td colspan="2">Body example
{
"numberOfPerson" : "3"
}</td><td>Response
{
"error": false,
"errormessage": "",
"newGroup": {
"numberOfPerson": 3,
"dateStart": "2023-09-06T07:20:00.605Z",
"dateFinish": null,
"ordersList": [],
"idRestaurant": "64e9f53c2fc327c9b3c97977",
"idRecipe": null,
"idTable": "64e9f9eb2fc327c9b3c97cd4",
"_id": "64f82820304ffe45e64ec920",
"__v": 0
}
}</td></tr>
<tr><td>DELETE</td><td>/restaurants/:idr/tables/:idt/group</td><td>Remove the customer group associated with a table</td></tr>
<tr><td colspan="3">{
"error": false,
"errormessage": "",
"newGroup": {
"_id": "64e9f9e12fc327c9b3c97ca6",
"tableNumber": "Tavolo 1",
"maxSeats": 10,
"group": null,
"restaurantId": "64e9f53c2fc327c9b3c97977",
"__v": 0
}
}</td></tr>
</table>

2.1.3.5 Orders of a customer group APIs

HTTP Method	Route	Description
GET	/restaurants/:idr/tables/:idt/group/orders	Retrieve orders within the customer group for a table

Response (for simplicity, I omit some codes)


```
{
"error": false,
"errormessage": "",
"orders": [
   {
   "_id": "64f82ddf304ffe45e64ec9d6",
   "idGroup": "64f82ddb304ffe45e64ec9bf",
   "idWaiter": "64e9f8622fc327c9b3c97b51",
   "items": [
      {
      "timeFinished": null,
      "idItem": "64e9f61d2fc327c9b3c979ae",
      "state": "notcompleted",
      "completedBy": null,
      "count": 1,
      "_id": "64f82ddf304ffe45e64ec9d7"
      }
    ],
    "state": "notStarted",
    "timeCompleted": null,
    "timeStarted": "2023-09-06T07:44:31.775Z",
    "type": "dish",
    "__v": 0
    }
]
}
```


POST	/restaurants/:idr/tables/:idt/group/orders	Create a new order within the customer group for a table

Body example

```
{
    "items" : [
        {"itemId" : "64d6a5e10884c1758a94dae9", "count" : "2"},
        {"itemId" : "64d6a5ee0884c1758a94daf3", "count" : "3"},
        {"itemId" : "64d6a5f60884c1758a94daf9", "count" : "1"}
    ]
}
```

PUT	/restaurants/:idr/tables/:idt/group/orders	Update an existing order within the customer group

Body examples

```
{"status" : "ready"}
{"status" : "served"}
```

PUT	/restaurants/:idr/tables/:idt/group/orders/:ido/items/:idi	Update an item within an order in the customer group

```
{"status" : "completed"}
```

2.1.3.6 Recipes of a customer group APIs

HTTP Method	Route	Description
GET	/restaurants/:idr/tables/:idt/group/recipe	Retrieve recipes associated with the customer group for a table

Response (for simplicity, I omit some codes)


```
{
   "error": false,
   "errormessage": "",
   "recipe": {
      "_id": "64e8be0f3e2717734e34bef1",
      "costAmount": 63.47,
      "dateOfPrinting": "2023-08-25T14:43:27.731Z",
      "idGroup": "64e8bddb3e2717734e34bdd0",
      "idCashier": "64e3805fd675814b2cfdca03",
      "itemsBought": [
        {
           "quantity": 1,
           "_id": "64e38199d675814b2cfdcb55"
        },
        {
           "quantity": 1,
           "_id": "64e38199d675814423fdcb5"
        }
      ],
    "__v": 0
   }
}
```


POST	/restaurants/:idr/tables/:idt/group/recipe	Caluclate a new recipe within the customer group for a table

2.1.3.7 Owner uses APIs

HTTP Method	Route	Description
GET	/restaurants/:idr/groups	Retrieve all customer groups for a restaurant
GET	/restaurants/:idr/recipe	Retrieve all recipes for a restaurant
GET	/restaurants/:idr/recipe/:idre	Retrieve a specific recipe for a restaurant
POST	/restaurants	Create a new restaurant
GET	/restaurants/:idr	Retrieve a specific restaurant by ID

I prefer not to add the snippets of code because are very similar to thouse above.

2.1.3.8 Self User APIs

HTTP Method	Route	Description
GET	/login	Log in a user

The Header of the request : 'authorization':'Basic '+ btoa("metiupaga8@gmail.com:admin"),


```
{
  "error": false,
  "errormessage": "",
  "token":
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJyZXN0YXVyYW50SWQiOiI2NGU5ZjUzYzJmYzMyN2M5YjNjOTc5NzciLCJ1c2VybmFtZSI6Ik1hdHRlbyIsInJvbGUiOiJvd25lciIsImVtYWlsIjoibWV0aXVwYWdhOEBnbWFpbC5jb20iLCJfaWQiOiI2NGU5ZjUzYzJmYzMyN2M5YjNjOTc5NzYiLCJpYXQiOjE2OTM5ODgyODUsImV4cCI6MTY5NDAwNjI4NX0.aJP-h4cbr87iw4W8t23mG-aNS4WRubeMRaV1EPVfCC0"
}
```


POST	/signup	Create a new user account

Body example

```
{
  "username": "Matteo",
  "email": "metiupaga8@gmail.com",
  "password": "admin",
  "restaurantName": "Matteo Restaurant"
}
```

GET	/users/:idu	Retrieve a specific user by ID

Response


```
{
  "error": false,
  "errormessage": "",
  "userDetails": {
  "_id": "64f82bbb304ffe45e64ec934",
  "recipesPrinted": [
    "64f82c0a304ffe45e64ec951"
  ],
  "username": "cashier2",
  "email": "email@cashier2.it",
  "role": "cashier",
  "idRestaurant": "64e9f53c2fc327c9b3c97977",
  "salt": "cfeb082ade2e6f91befd3e63cd769fa5",
  "digest":
"6425d4fa3896e8356bd9ec7e586c859713a8d0a7da55d78a5288f2b4cc7756f66dd86f88a0cfe12fa4a6d1b9b883cd3d4867b7567830b630cdbb4d689053f2fe",
"__v": 1
}
}
```


PUT	/users/:idu	Update the password for a specific user's by ID

Body example

```
{
 "passwordToChange" : "admin",
 "newPassword" : "adminNew"
}
```

2.1.4 Authentication Management

For reaching all those Endpoints, every request must been at least authenticated. This implies that there is a way that a client can authenticate. The `"/login"` API is organized for doing that.

2.1.4.1 Login API

The Login API in the server is structured in this way:

```
app.get('/login', MW.basicAuthentication, EP.login)
```

How we can see there is only one middleware called basicCauthentication and an endpoint called login.

2.1.4.2 basicAuthentication

The basicAuthentication is like this:

```
export const basicAuthentication = passport.authenticate("basic", {
  session: false,
});
```

and the passport.authenticate uses a basic strategy for the login. It's simply a middleware that when is invoked, run this async function:

```
passport.use(
  new passportHTTP.BasicStrategy(
       async function (username: string,password: string,done: Function) {
         let user: User.User = await User.UserModel.findOne({ email: username });
         if (!user) {
            return done({ statusCode: 401, message: "Invalid credentials" }, false);
         }
         if (user.isPasswordCorrect(password)) {
         switch (user.role) {
            case "owner":
            user = new Owner.OwnerModel(user);
            break;
         case "bartender":
            user = new Bartender.BartenderModel(user);
            break;
         case "cashier":
            user = new Cashier.CashierModel(user);
            break;
         case "cook":
            user = new Cooker.CookModel(user);
            break;
         case "waiter":
            user = new Waiter.WaiterModel(user);
            break;
         }
         return done(null, user);
         } else {
            return done({ statusCode: 401, message: "Invalid credentials" }, false);
         }
    })
);
```

How we can see it requires three arguments, an username a password and a calback function named done, that is the way that permit to reach the endpoint in the chain.

When a client request to the server the API "/login", with the header formatted like this `'authorization' : 'Basic ' + btoa( <email> + ':' + <password>)`,
basicAuthentication middleware checks the header of the request, and then, if contains the basic authentication, so the request is correct, invoke the async function.
Then, the ascync function checks If the username and the password are correct, with a mid step that imply a data retrieve from the database, and if it is, the middleware with `return done(null, user);` leave the command at the endpoint EP.login.

Else, if the credentials are invalids, it returns an error to the error's endpoint.

2.1.4.3 login

The login endpoint works like this:

```
export function login(req: Request, res: Response, next: NextFunction) {
  // If it's reached this point, req.user has been injected.

  const authenticatedUser: any = new User.UserModel(req.user);

  var token;

  if (authenticatedUser.role === "owner") {
    token = {
      restaurantId: authenticatedUser.restaurantOwn,
      username: authenticatedUser.username,
      role: authenticatedUser.role,
      email: authenticatedUser.email,
      _id: authenticatedUser._id,
    };
  } else {
    token = {
      restaurantId: authenticatedUser.idRestaurant,
      username: authenticatedUser.username,
      role: authenticatedUser.role,
      email: authenticatedUser.email,
      _id: authenticatedUser._id,
    };
  }

  const secret = process.env.JWT_SECRET;

  if (!secret) {
    throw new Error("JWT secret is not defined");
  }

  const options = {
    expiresIn: "5h",
  };
  const tokenSigned = jsonwebtoken.sign(token, secret, options);

  return res
    .status(200)
    .json({ error: false, errormessage: "", token: tokenSigned });
}
```

If the control flow reaches this point, it means that the property req.user is been injected by the BasicAuthentication middleware. Now, how we can see in the snippet's code, the login endpoints retrieve from a file named `.env,` stored in the backend, a string containing the JWT_SECRET.
This is a secret string that are used by the `jsonwebtoken.sign()` function to build and sign a JSON Web Token. After the sign the token must be resent to the client.

What is a JSON Web Token?
A JSON Web Token is composed by three parts:

1) The header, that usually contains the algorithm's sign for generate the token, coded in base 64.
2) The payload, that contains useful information to store and the UNIX timestamp of the expiring time, coded in base 64.
3) Sign, this is generated by an algorithm that takes in input the combination of the header and the payload in base 64, `base64UrlEncode(header) + "." + base64UrlEncode(payload)`, and the secret.

2.1.4.4 Example of a JWT in the project

For example the JWT:
`eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.`
`eyJyZXN0YXVyYW50SWQiOiI2NGU5ZjUzYzJmYzMyN2M5YjNjOTc5NzciLCJ1c2VybmFtZSI6Ik1hdHRlbyIsInJvbGUiOiJvd25lciIsImVtYWlsIjoibWV0aXVwYWdhOEBnbWFpbC5jb20iLCJfaWQiOiI2NGU5ZjUzYzJmYzMyN2M5YjNjOTc5NzYiLCJpYXQiOjE2OTM5MzA1NzEsImV4cCI6MTY5Mzk0ODU3MX0.`
`ITKLcB03tyZUcyPtmwNTRdkzcXYX8UttkqgSZlrlROI`

Is like that decoded:

```
Header
{
  "alg": "HS256",
  "typ": "JWT"
}
```

```
Payload
{
  "restaurantId": "64e9f53c2fc327c9b3c97977",
  "username": "Matteo",
  "role": "owner",
  "email": "metiupaga8@gmail.com",
  "_id": "64e9f53c2fc327c9b3c97976",
  "iat": 1693930571,
  "exp": 1693948571
}
```

```
Signature
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  mysecretpassword
)
```

Obviously the secret for sign the JSON Web Token must be store in a safe manner.

2.1.4.5 Frontend

Now that the client has the JWT, It must be store in the local storage, and then reutilized each times a request is sent.

For example if the client now want to access to the `PUT /users/:idu` Express's API, must be put in the header a special string composed like this: `Authorization',`Bearer ${accessToken}``

2.1.4.6 Backend

For each endpoint, except the login, there is a special middleware called verifyJWT. The only task for it is to verify if the token has been manipulated by someone.

```
export const verifyJWT = jwt({
  secret: process.env.JWT_SECRET,
  algorithms: ["HS256"],
});

app.put('/users/:idu', MW.verifyJWT, MW.isThatUser, EP.modifyPassword)
```

If the token is correct, the chain of middleware can continuate, else will be catch an error.

2.1.5 Generals opinions

In the personal opinion section, I would like to say that for every request to the exposed APIs, aimed at reaching the actual endpoint responsible for making changes to the database, the server analyzes the user and verifies certain conditions and essential permissions for the proper functioning of the application.

For example, if an owner makes a request to '/restaurants/:idr/cooks,' it must first pass through several checks such as 'verifyJWT,' 'isOwner,' 'isOwnerOfThisRestaurant,' and 'isUserAlreadyExist' before reaching the 'createCookAndAddToARestaurant' endpoint. This ensures strong database-level consistency for the application.

Although the use of middleware chains may lead to a decrease in performance, it makes the application more modular and easier for me to maintain.

2.2 Database

The Database component is where the application's data is stored. This component can communicate only with the Backend component.

2.2.1 MongoDB

The Database used for developing this application is MongoDB. It is a Database designed to store, retrieve and manage data in a format similar to the JSON documents, and this choice fits very well with Express Server. Each document belongs to one collection.

Although usually using a document-oriented database like MongoDB doesn't imply a fixed schema like the relational databases, I decide to use it likes a relational database, doing reference to the key of a single object. I made this decision to perform data consistency.

2.2.2 Relational Model

This link shows the relational model of the database. (I prefer to set a link to the github folder for a better view)
How we can see there are 7 main collection. All the documents stored in each collection have a id of type ObjectId, used from the application for retrieving them from the database.
Differently than requested, each person that subscribe to the application can create their own restaurant. This is the main change applied by myself to the application.

2.2.3 Description of each Collection

Now I want to discuss about the property that collections have. This is the Model's folder stored in Github.

2.2.3.1 Restaurant

The restaurant collection contains

- **restaurantName**: This field stores the name of the restaurant. It is of type String and is required, meaning that every restaurant document must have a name.
- **ownerId**: This field stores the unique identifier of the owner of the restaurant. It is of type ObjectId and is required. It also references the "Owner" model, establishing a relationship between the restaurant and its owner.
- **tables**: This field is an array that stores the unique identifiers of tables within the restaurant. It is of type ObjectId and is not required. The array references to "Table" collection.
- **items**: Similar to tables, this field is an array that stores the unique identifiers of items served by the restaurant. It is of type ObjectId and is not required. The array references to "Item" documents.
- **cookers**: This field is an array that stores the unique identifiers of cooks working at the restaurant. It is of type ObjectId and is not required. The array references to "Cook" collection.
- **waiters**: This field is an array that stores the unique identifiers of waiters employed at the restaurant. It is of type ObjectId and is not required. The array references to "Waiter" collection.
- **cashiers**: Similar to cookers and waiters, this field is an array that stores the unique identifiers of cashiers working at the restaurant. It is of type ObjectId and is not required. The array references to "Cashier" collection.
- **bartenders**: This field is an array that stores the unique identifiers of bartenders employed at the restaurant. It is of type ObjectId and is not required. The array references to "Bartender" collection.
- **recipes**: This field is an array that stores the unique identifiers of recipes present in the restaurant. It is of type ObjectId and is not required. The array references to "Recipe" collection.
- **groups**: Similar to other fields, this field is an array that stores the unique identifiers of groups associated with the restaurant. It is of type ObjectId and is not required. The array references to "Group" collection.

This is the schema model inside the Express Server

2.2.3.2 Table

The Table collection contains:

- **tableNumber**: This field stores the unique identifier or number assigned to the table. It is of type String and is required, ensuring that each table document has a unique identifier.
- **maxSeats**: This field represents the maximum number of seats available at the table. It is of type Number and is required, indicating the maximum capacity of the table in terms of seating.
- **group**: This field stores the unique identifier of the group associated with the table, if applicable. It is of type ObjectId and is not required. The field can contain a reference to a "Group" document, allowing tables to be associated with specific groups, one by one.
- **restaurantId**: This field stores the unique identifier of the restaurant to which the table belongs. It is of type ObjectId and is required. The field contains a reference to a "Restaurant" document, establishing a relationship between the table and its restaurant.

This is the schema model inside the Express Server

2.2.3.3 Item

The Item collection contains:

- **itemName**: This field stores the name of the menu item. It is of type String, required, and unique. The uniqueness ensures that each menu item has a distinct name within the restaurant.
- **itemType**: This field represents the type or category of the menu item. It is of type String and is required. It typically accepts values from an enum called ItemType, which restricts the item type to “dish” or “drink”
- **price**: This field holds the price of the menu item. It is of type Number and is required. It indicates the cost of the menu item.
- **preparationTime**: This field specifies the time required to prepare the menu item. It is of type Number and is required. It represents the estimated preparation time in minutes.
- **idRestaurant**: This field stores the unique identifier of the restaurant to which the menu item belongs. It is of type ObjectId and is required. This field establishes a relationship between the menu item and its parent restaurant using references.
- **countServered**: This field represents the number of times the menu item has been ordered. It is of type Number and is required. It keeps track of how frequently the menu item is ordered.

This is the schema model inside the Express Server

2.2.3.4 Group

The Group collection contains:

- **numberOfPerson**: This field stores the number of people in the group. It is of type Number and is required. It indicates the size of the group.
- **dateStart**: This field represents the start date and time of the group's reservation or visit. It is of type Date and is required. It marks the beginning of the group's interaction with the restaurant.
- **ordersList**: This field is an array that holds references to orders associated with the group. It is of type ObjectId and is not required. Each element in the array is a reference to the order model.
- **idRestaurant**: This field stores the unique identifier of the restaurant to which the group's reservation is related. It is of type ObjectId and is not required. This field establishes a relationship between the group and the restaurant using references.
- **idRecipe**: This field stores the unique identifier of a recipe associated with the group, if applicable. It is of type ObjectId and is not required. This field establishes a relationship between the group and a recipe using references.
- **idTable**: This field stores the unique identifier of a table where the group is seated, if applicable. It is of type ObjectId and is not required. This field establishes a relationship between the group and a table using references.

This is the schema model inside the Express Server

2.2.3.5 Recipe

The Recipe collection contains:

- **costAmount**: This field stores the cost or total amount of the recipe. It is of type Number and is required. It represents the total cost associated with the items bought as part of this recipe.
- **dateOfPrinting**: This field represents the date when the recipe was printed or generated. It is of type Date and is required. It indicates when the recipe was created.
- **idGroup**: This field stores the unique identifier of the group associated with the recipe. It is of type ObjectId and is required. This field establishes a relationship between the recipe and a group (referring to the "Group" model) and indicates which group the recipe belongs to.

- **idCashier**: This field stores the unique identifier of the cashier who generated the recipe. It is of type ObjectId and is required. This field establishes a relationship between the recipe and a cashier (referring to the "Cashier" model) and indicates which cashier created the recipe.
- **itemsBought**: This field is an array that holds information about the items bought as part of the recipe. Each element in the array includes:
    - **quantity**: This field represents the quantity of a specific item bought in the recipe. It is of type Number and is required.
    - **_id**: This field stores the unique identifier of the item bought. It is of type ObjectId and is required. This field establishes a relationship between the recipe and the items (referring to the "Item" model) that were purchased.

This is the schema model inside the Express Server

2.2.3.6 Order

The order collection contains:

- **idGroup**: This field stores the unique identifier of the group to which the order belongs. It is of type ObjectId and is required. It establishes a relationship between the order and a group (referring to the "Group" model) and indicates which group the order is associated with.
- **idWaiter**: This field stores the unique identifier of the waiter who took the order. It is of type ObjectId and is required. It establishes a relationship between the order and a waiter (referring to the "Waiter" model) and indicates which waiter handled the order.
- **items**: This field is an array that contains information about the items included in the order. Each element in the array includes the following subfields:
    - **timeFinished**: This field represents the time when the item was finished, and it is of type Date. It is optional and is used to track when an item is completed.
    - **idItem**: This field stores the unique identifier of the item included in the order. It is of type ObjectId and is required. It establishes a relationship between the order and the items (referring to the "Item" model) included in the order.
    - **state**: This field represents the state of the item within the order. It is of type String and must be one of the predefined values in the StateItem enum. It is required and indicates whether the item is completed or not.
    - **completedBy**: This field stores the unique identifier of the cook or bartender who completed the item. It is of type ObjectId and is optional. It is used to track who completed the item, especially if it's in a "completed" state.
    - **count**: This field represents the quantity of the item included in the order. It is of type Number and is required.
- **state**: This field represents the overall state of the order. It is of type String and must be one of the predefined values in the StateOrder enum. It is required and indicates the status of the order, such as "completed" or "notcompleted."
- **timeCompleted**: This field stores the time when the order was completed. It is of type Date and is optional. It is used to track the completion time of the order.
- **timeStarted**: This field represents the time when the order was started. It is of type Date and is required. It indicates when the order was initiated.
- **type**: This field represents the type of the order. It is of type String and must be one of the predefined values in the ItemType enum. It is required and specifies the type of items included in the order, such as "dish" or "drink."

This is the schema model inside the Express Server

2.2.3.7 User

The user collection contains

- **username**: A required field that stores the user's username, which must be a string.
- **email**: A required field that stores the user's email address, which must be a string.

- **digest**: A required field that stores the user's password digest, which is an hashed version of their password.
- **role**: A required field that specifies the user's role, and it's limited to a set of predefined values defined by the RoleType enum.
- **salt**: A required field that stores a unique value used in the hashing process of the user's password.
- **idRestaurant**: References a "Restaurant" model using its ObjectId. This field allows associating a user with a specific restaurant.

This is the schema model inside the Express Server
In addition of the user collection, there are a few schema, the various kind of users, that extends all the property of the User collection. This is useful when I had to works with many kind of users.

2.2.3.8 Cook Schema

The cookSchema describes the structure of the "Cook" model in the application. Here's a description of the additional field in the schema:

- **itemsPrepared**: A required field that represents the items prepared by the cook. It's an array of objects, each containing two properties:
    - - **idItem**: This is a reference to an "Item" model using its ObjectId. It specifies the item that the cook has prepared.
    - - **count**: This field stores the quantity of the item prepared by the cook, and it must be a number.

This is the schema model inside the Express Server

2.2.3.9 Bartender Schema

The bartenderSchema describes the structure of the "Bartender" model in the application. Here's a description of the additional field in the schema:

- **itemsPrepared**: A required field that represents the items prepared by the bartender. It's an array of objects, each containing two properties:
    - **idItem**: This is a reference to an "Item" model using its ObjectId. It specifies the item that the bartender has prepared.
    - **count**: This field stores the quantity of the item prepared by the bartender, and it must be a number.

This is the schema model inside the Express Server

2.2.3.10 Cashier Schema

The bartenderSchema describes the structure of the "Cashier" model in the application. Here's a description of the additional field in the schema:

- **recipesPrinted**: This field represents an array of ObjectId references to "Recipe" documents. It is required and indicates the restaurants for which the cashier has printed recipes.

This is the schema model inside the Express Server

2.2.3.11 Waiter Schema

The waiterSchema describes the structure of the "Waiter" model in your application. Here's a description of the fields in the schema:

- **ordersAwaiting**: This field represents an array of ObjectId references to "Order" documents. It is required and indicates the orders that are awaiting service by the waiter.
- **ordersServed**: This field also represents an array of ObjectId references to "Order" documents. It is required and indicates the orders that have been served by the waiter.

This is the schema model inside the Express Server

2.2.3.12 Owner Schema

The ownerSchema describes the structure of the "Owner" model in the application. Here's a description of the fields in the schema:

- **restaurantOwn**: This field represents a single ObjectId reference to a "Restaurant" document. It is not required (as indicated by required: false) and is used to associate an owner with a restaurant they own.

This is the schema model inside the Express Server

2.2.4 Generals options

As for the database section, since the choice of using a relational paradigm for the MongoDB database involves the fact that many operations are performed across multiple collections, and MongoDB guarantees transaction atomicity only at the document level, this type of solution may not ensure the integrity of the database. To solve this issue, given that the application involves multi-collection operations, one feature to consider adding would be to add a MongoDB replica set and to utilize the atomic operations introduced in MongoDB version 7.0. This would enhance the overall of data integrity and consistency within the application.

2.3 Frontend

The Frontend component is the visualization part of the application. It can only communicate with the Backend component for retrieving data.

2.3.1 Angular

For developing the Frontend component I use the Angular framework. In this paragraph I want to focalize in 4 main part of the application:

1) Services
2) Components
3) Guards
4) Routes

This is the Angular's folder in the Github repository.

2.3.1.1 Services

For the communication with the Backend component, I decide to divide each type of HTTP Request in separate services.

Each services that implies communication with the Backend, i.e makes HTTP Request, imports in the constructor two services explaned down here:

Service Name	Methods	Description
JwtService	getToken, setToken	This service has the task to store and return the Java Web Token, retrieved by the login method.
UserPropertyService	getRule, getRestaurant, getEmail, getUsername, isLogged, logout	This service has the task to retrieve many informations stored in the Java Web Token.

2.3.1.2 Auth Services

Service Name	Methods	Parameters
AuthRequestService	login	This service makes possible the communication with the login API.
SignupRequestService	signup	This service makes possible the communication with the signup API.

2.3.1.3 General Services for the communication with the backend

In addition to importing JwtService and UserPropertyService in the constructor, all these services listed below extend a particular class which has the task of constructing the header. This specific class is called AuthenticatedRequest, and it overrides a special method called create_options extended with the BaseRequest abstract class, adding the Bearer Token.

GroupsRequestService

Method	Description
getGroupFromTable	Retrieves the customer group associated with a table
removeGroupFromTable	Removes a customer group from a table
addGroupToTable	Adds a customer group to a table
getGroups	Retrieves a list of customer groups

ItemsRequestService

Method	Description
addItem	Adds a new menu item
getItems	Fetches a list of menu items
deleteItem	Deletes a menu item

OrdersRequestService

Method	Description
getOrdersByTable	Retrieves orders for a specific table
getOrdersDishNotStartedByTable	Retrieves dish orders that have not started preparation for a specific table

getOrdersDrinkNotStartedByTable	Retrieves drink orders that have not started preparation for a specific table
createGroupOrder	Creates a new group order
modifyItemOfOrderCompleted	Modifies an item in an order to mark it as completed.
modifyOrderReady	Modifies the status of an order to indicate it's ready.
modifyOrderServed	Modifies the status of an order to indicate it's served.

RecipesRequestService

Method	Description
calculateRecipe	Calculates a recipe.
getRecipes	Fetches a list of recipes.
getRecipe	Retrieves a specific recipe.

RestaurantRequestService

Method	Description
createRestaurant	Creates a new restaurant.

SelfUserRequestService

Method	Description
getMySelf	Retrieves information about the currently authenticated user.
modifyPassword	Modifies the user's password.

TablesRequestService

Method	Description
getTables	Fetches a list of restaurant tables.
deleteTable	Deletes a restaurant table.
addTable	Adds a new restaurant table.
getTablesNotEmpty	Retrieves tables that are not empty.

UsersRequestService

Method	Description
get_cashiers	Retrieves a list of cashiers.

get_waiters	Retrieves a list of waiters.
delete_cashier	Deletes a cashier.
delete_waiter	Deletes a waiter.
get_cooks	Retrieves a list of cooks.
get_bartenders	Retrieves a list of bartenders.
delete_cook	Deletes a cook
delete_bartender	Deletes a bartender.
create_user	Creates a new user.

2.3.1.4 Service for the creation of the PDF recipe

This is a special service I added for the creation of a PDF for a specific recipe.

PdfGeneratorService

Method	Description
generateReceipt	It create and return a pdf with the recipe associate when the “calculate recipe” button is pressed.

2.3.1.5 Services for the Real Time Communication

There is a small separate discussion to be made for the real time communication of the application; Every time a client launches a feature of the application, the dedicated component connects to the socket.io server by entering a particular room whose id is the restaurant's id, takes the data relating to the feature through an http request, and every time an update of this data occurs, the relevant change is forwarded to each client connected to that room, so that it can be shown on the screen.

Obviously to maintain data persistence, the client that performs the update makes both an http request to modify the data in the database and sends a socket.io event to the server.

An example of that is the waiter’s orders awaiting component

```
constructor(
    private ups: UserPropertyService,
    private socketService: SocketService,
    private ors: OrdersRequestService,
    private srus: SelfUserRequestService
  ) {
    this.get_myself();

    this.socketService.joinRestaurantRoom(
      this.ups.getRestaurant() + ups.getId() + 'ordersAwaited'
    );
    const socket = socketService.getSocket();

    socket.fromEvent('fetchOrderReady').subscribe((data: any) => {
      console.log('fetchOrderReady');

      const index = this.ordersAwaiting.findIndex(
```

```
        (order) => order._id === data.order._id
      );

      if (index !== -1) {
        this.ordersAwaiting[index] = data.order;

      } else {
        console.log('Ordine non trovato');
      }
    });
  }
```

How we can see this snippet of code is related to the constructor function of the DisplayOrdersAwaiting component, and what it's makes is:

1) retrieve, with the function get_myself(), the various orders awaited;
2) listens for any updates sents by the server.

From the cooker side happen this:

```
orderCompleted(tableId: string, orderId: string) {
    this.ors.modifyOrderReady(tableId, orderId).subscribe((data) => {
      const orderIndexToRemove = this.orders.findIndex((orderObj) => orderObj._id === orderId);
      if (orderIndexToRemove !== -1) {
        this.orders.splice(orderIndexToRemove, 1);
        console.log(data)
        this.socketService.emitOrderDishCompleted(this.ups.getRestaurant(), data.orderModifyied);
      } else {
        console.log("Ordine non trovato");
      }
    }
   )
}
```

```
When a order is completed, the function
this.socketService.emitOrderDishCompleted(this.ups.getRestaurant(), data.orderModifyied); sents
event to the server
```

And this is what happen on the server side

```
socket.on('setOrderDishStatus', (room, order) => {
      socket.broadcast.to(room).emit('fetchOrderDishStatus', {order : order});
      const str = room + order.idWaiter + "ordersAwaited"
      io.to(room + order.idWaiter + "ordersAwaited").emit('fetchOrderReady', {order : order});
    });
```

```
The function
socket.broadcast.to(room).emit('fetchOrderDishStatus', {order : order})
sends the order status to all the cooks linked to this room.
```

```
Else the function
io.to(room + order.idWaiter + "ordersAwaited").emit('fetchOrderReady', {order : order})
sends the updates to the waiter associated to this order.
```

This type of pattern makes both real-time communication and data persistence in the database very light.

Services

These methods allow the Angular application to communicate with the backend via socket.io and perform various actions related to restaurants, tables, menu items, customer groups, orders, recipes, and order status updates.

Method	Description
joinRestaurantRoom	Joins a socket.io room associated with a specific restaurant.
emitFetchTable	Emits a request to fetch tables for a specific restaurant.
emitFetchItems	Emits a request to fetch menu items for a specific restaurant.
emitFetchGroups	Emits a request to fetch customer groups for a specific restaurant.
emitFetchOrders	Emits a request to fetch orders for a specific restaurant.
emitNewOrderDrink	Emits a new drink order for a specific restaurant and table.
emitNewOrderDish	Emits a new dish order for a specific restaurant and table.
emitItemOfOrderDishStatus	Emits a status update for an item in a dish order.
emitItemOfOrderDrinkStatus	Emits a status update for an item in a drink order.
emitOrderDrinkCompleted	Emits a notification that a drink order is completed.
emitOrderDishCompleted	Emits a notification that a dish order is completed.
emitFetchRecipes	Emits a request to fetch recipes for a specific restaurant.
getSocket	Returns the socket instance for further customization or event handling.

2.3.2 Components

The structure of the project are composed of two components type, the Authentication component and the User components.

2.3.2.1 Authentication

This part here has the task of registering and logging the various kind of users in the application.
Below I show the main components fo the Authentication and a specific description of them.

Component’s name	Description
AuthenticationComponent	This component acts as a parent to the login and registry components. Infact it shows a the login or the register components, switching them by a simple button

LoginUserComponent	This component shows the login page for authenticating a client, It requires an email and a password.
SignupUserComponent	This component shows the singup page for registering a client, It requires an username, an email, a restaurant name and a password.

Those component's above form an angular module, and the routes for reaching them from the **relative path** of the module are those:

Componet's name	Relative Path
AuthenticationComponent	/
LoginUserComponent	/login
SignupUserComponent	/signup

For access the Authentication Component and the other two sub components above, there are one guard that checks if the user are already logged or not.

Guard's name	Description
isAuthGuard	This guard has the role of checks if the user, in the current session, is logged or not, if is logged It can't reach the Authentication component, else yes.

2.3.2.2 Users

This part here show the main content of the application, and I decide to divide it into 5 parts, the various kinds of users, and one shared between them, the Manage-Account component. Each parts form a module.

Bartender's Module

Component's name	Description
BartenderComponent	This component acts as a parent to all the features of the bartender.
DisplayQueueComponent	This component shows the drink's orders queue for that particular restaurant.
ManageAccountComponent	This component is used for change the password and visualize the user's details

Componet's name	Relative Path
BartenderComponent	/
DisplayQueueComponent	/queue
ManageAccountComponent	/manage-account

Cooker's Module

Component's name	Description

CookComponent	This component acts as a parent to all the features of the cook.
DisplayQueueComponent	This component shows the dish's orders queue for that particular restaurant.
ManageAccountComponent	This component is used for change the password and visualize the user's details

Componet's name	Relative Path
AuthenticationComponent	/
DisplayQueueComponent	/queue
ManageAccountComponent	/manage-account

Cashier's Module

Component's name	Description
CashierComponent	This component acts as a parent to all the features of the cashier.
DisplayTablesComponent	The DisplayTablesComponent is used for displaying and managing restaurant tables information.
VisualizeTableDetailsComponent	It's responsible for displaying detailed information about a specific table, and can be computated the bill.
ManageAccountComponent	This component is used for change the password and visualize the user's details

Componet's name	Relative Path
AuthenticationComponent	/
DisplayTablesComponent	/tables
VisualizeTableDetailsComponent	/tables/:id
ManageAccountComponent	/manage-account

Waiter's Module

Component's name	Description
CashierComponent	It serves as the main component for waiters in the restaurant management system
DisplayTablesComponent	It is responsible for displaying information related to restaurant tables.
DisplayOrdersAwaitingComponent	It displays a list of orders that are awaiting preparation.

Component's name	Description
CashierComponent	It serves as the main component for waiters in the restaurant management system
DisplayOrdersServedComponent	It shows a list of orders that have been served to customers.
CreateOrderComponent:	It allows waiters to create new orders for specific tables.
ManageAccountComponent	This component is used for change the password and visualize the user's details

Component's name	Relative Path
CashierComponent	/
DisplayTablesComponent	/tables
DisplayOrdersAwaitingComponent	/orders-awaiting
DisplayOrdersServedComponent	/orders-served
CreateOrderComponent:	/tables/:id
ManageAccountComponent	/manage-account

Owner's Module

Component's name	Description
OwnerComponent	It serves as the main component for waiters in the restaurant management system
DisplayUsersComponent	It's responsible for displaying a list of users associated to the restaurant
CreateUserComponent	It's used for creating new employee within the restaurant management system.
DisplayItemsComponent	It's responsible for displaying a list of menu items offered by the restaurant.
CreateItemComponent	It's designed to create a new items to be offered by the restaurant.
DisplayTablesComponent:	It provides a user interface for visualizing and managing restaurant tables
CreateTableComponent	It's used for creating and configuring new tables within the restaurant.
DisplayCustomersComponent	It displays a detailed history of groups eating in the restaurant.
VisualizeCustomerDetailsComponent	It allows owner to view detailed information about a specific customer group.
DisplayBartendersDataComponent	It provides data and insights related to bartender performance
DisplayWaitersDataComponent	It provides data and insights related to waiter performance.

DisplayCooksDataComponent	It provides data and insights related to cook performance
DisplayCashiersDataComponent	It provides data and insights related to cashier performance.
VisualizeItemsDataComponent	It allows to visualize data and analytics related to menu items
VisualizeCustomersDataComponent	It provides insights into customer behavior and preferences.
ManageAccountComponent	This component is used for change the password and visualize the user’s details

Component’s name	Relative Path
OwnerComponent	/
DisplayUsersComponent	/users/visualize
CreateUserComponent	/users/create
DisplayItemsComponent	/items/visualize
CreateItemComponent	/items/create
DisplayTablesComponent:	/tables/visualize
CreateTableComponent	/tables/create
DisplayCustomersComponent	/groups/visualize
VisualizeCustomerDetailsComponent	/groups/:id
DisplayBartendersDataComponent	/data-analytics/bartenders
DisplayWaitersDataComponent	/data-analytics/waiters
DisplayCooksDataComponent	/data-analytics/cooks
DisplayCashiersDataComponent	/data-analytics/cashiers
VisualizeItemsDataComponent	/data-analytics/items
VisualizeCustomersDataComponent	/data-analytics/customers
ManageAccountComponent	/manage-account

2.3.2.3 Main Component

The main components acts like a container for the visualization of the various features, including the Authentication's and the User's components.
The main component is utilized for charge the modules of the various component descripted above.

Component's name	Description
AppComponent	The AppComponent is the root component of the Angular application. It serves as the container for all other components and provides the overall structure and layout for the entire application.
PermissionDeniedComponent	This route loads the Permission Denied Component, which is displayed when a user attempts to access a route or feature for which they don't have the required permissions or roles.

These routes help organize and secure access to different parts of the restaurant management system based on user roles and permissions. Acces a particular route charge in a dynamic way the Module associated with it for then construct the absolute path.

Module charged	Relative Path	Guard	User Permitted
OwnerModule	/owner-dashboard	authGuard, roleGuard, roleChildGuard	Owner
BartenderModule	/bartender-dashboard	authGuard, roleGuard, roleChildGuard	Bartender
CookModule	/cooker-dashboard	authGuard, roleGuard, roleChildGuard	Cook
WaiterModule	/waiter-dashboard	authGuard, roleGuard, roleChildGuard	Waiter
CashierModule	/cashier-dashboard	authGuard, roleGuard, roleChildGuard	Cashier
AuthenticationModule	/authentication		
PermissionDeniedComponent	/permission-denied		
AuthenticationModule	/**		

2.3.3 Guards

How we can see on the table above there is 3 types of guards that protect the various routes:

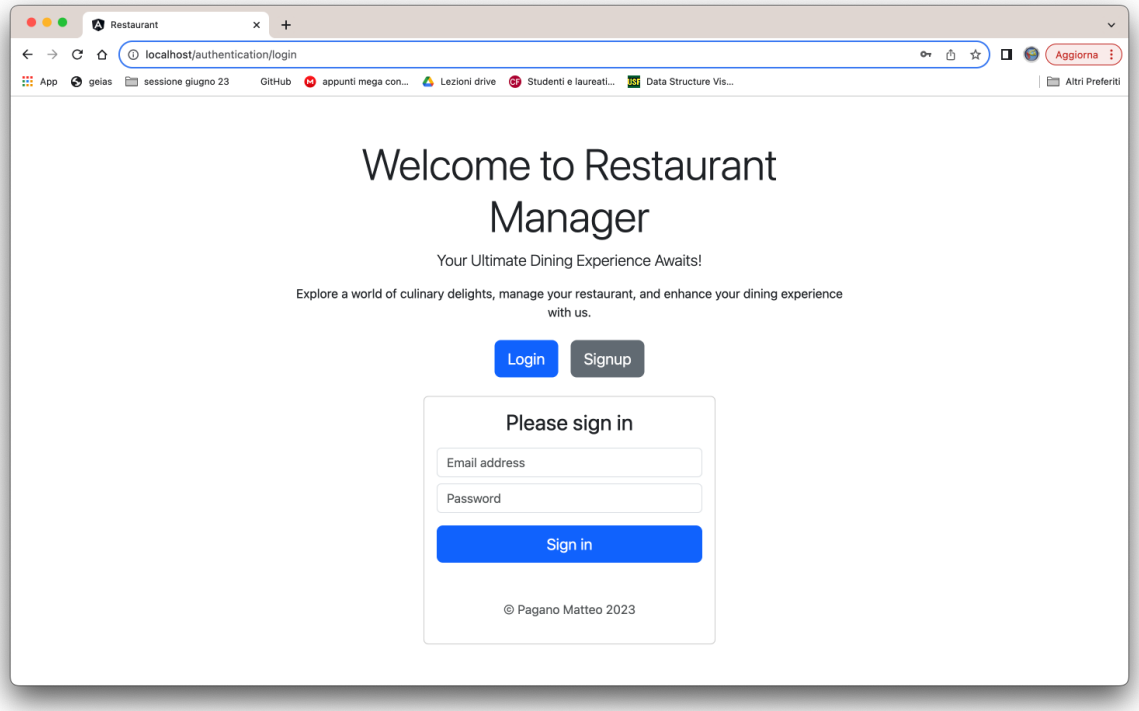
- authGuard: Ensures that a user is authenticated (logged in) before granting access to certain routes. If a user is not authenticated, they are redirected to the login page.
- roleGuard: Verifies that a user has the required role (e.g., owner, bartender, cook, waiter, cashier) to access specific routes. Unauthorized users are redirected to a permission-denied page.
- roleChildGuard: Similar to roleGuard, but it specifically guards child routes within role-based modules to ensure users with the correct role can access child routes.

3 Overview for different roles usages

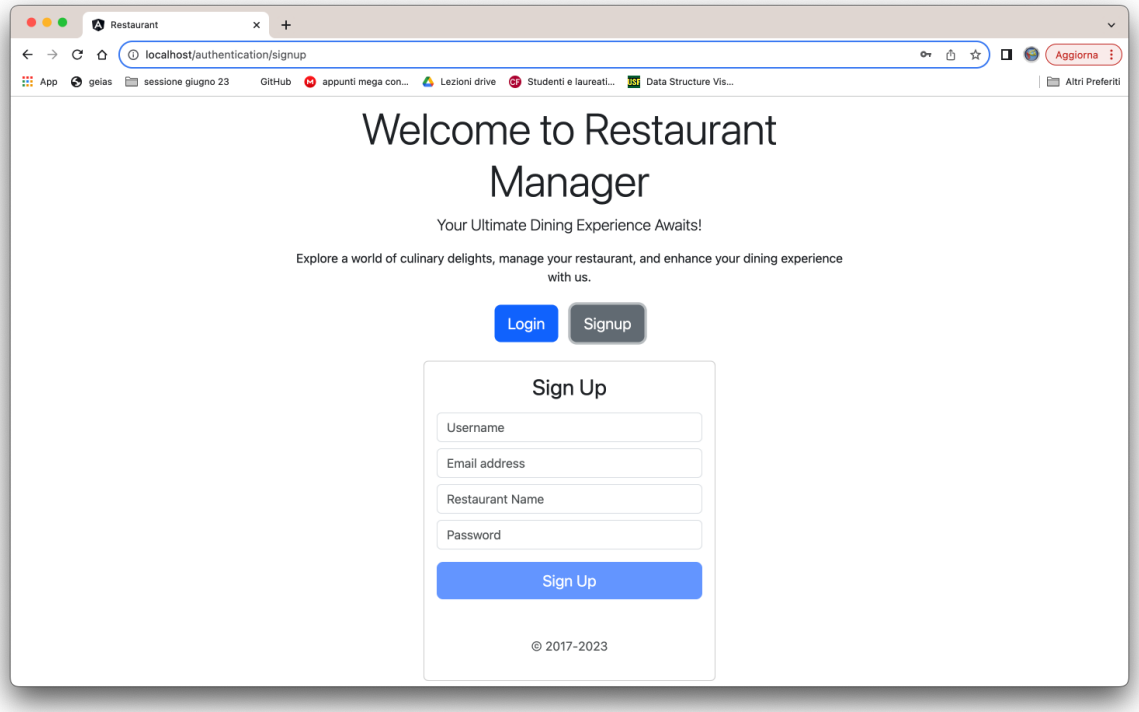
In this last paragraph I want to share some examples of grafich interface of the application

3.1 Authentication Interface

3.1.1 Login

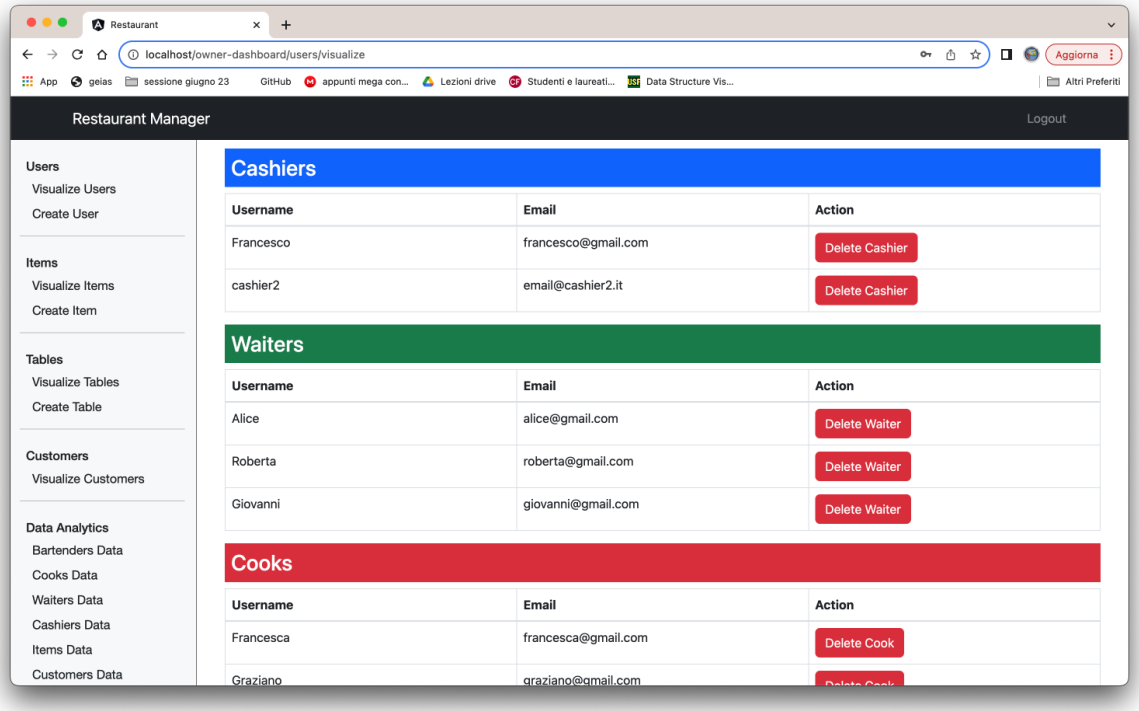


3.1.2 Signup

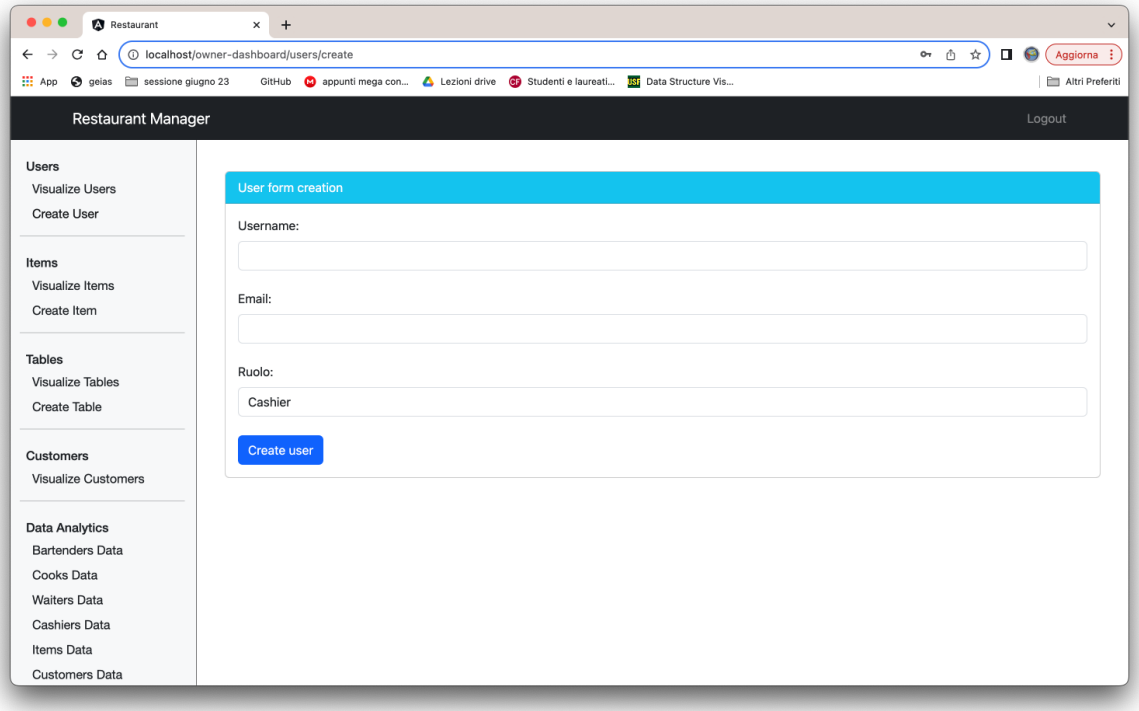


3.2 Owner

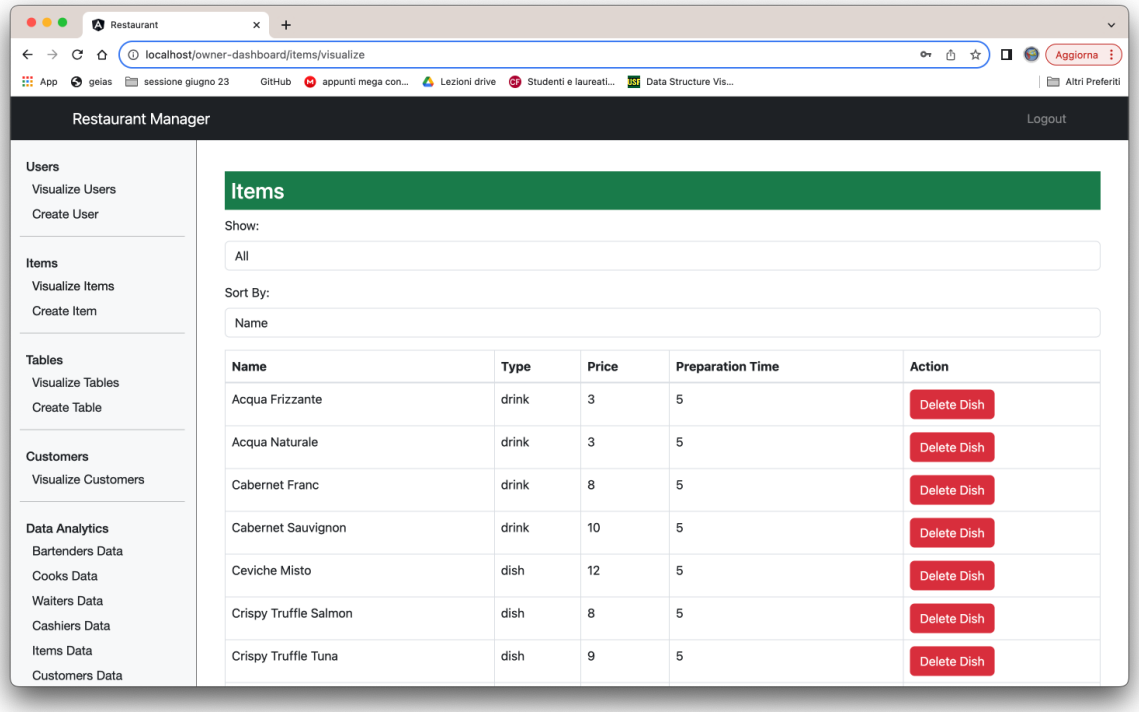
3.2.1 Visualize User



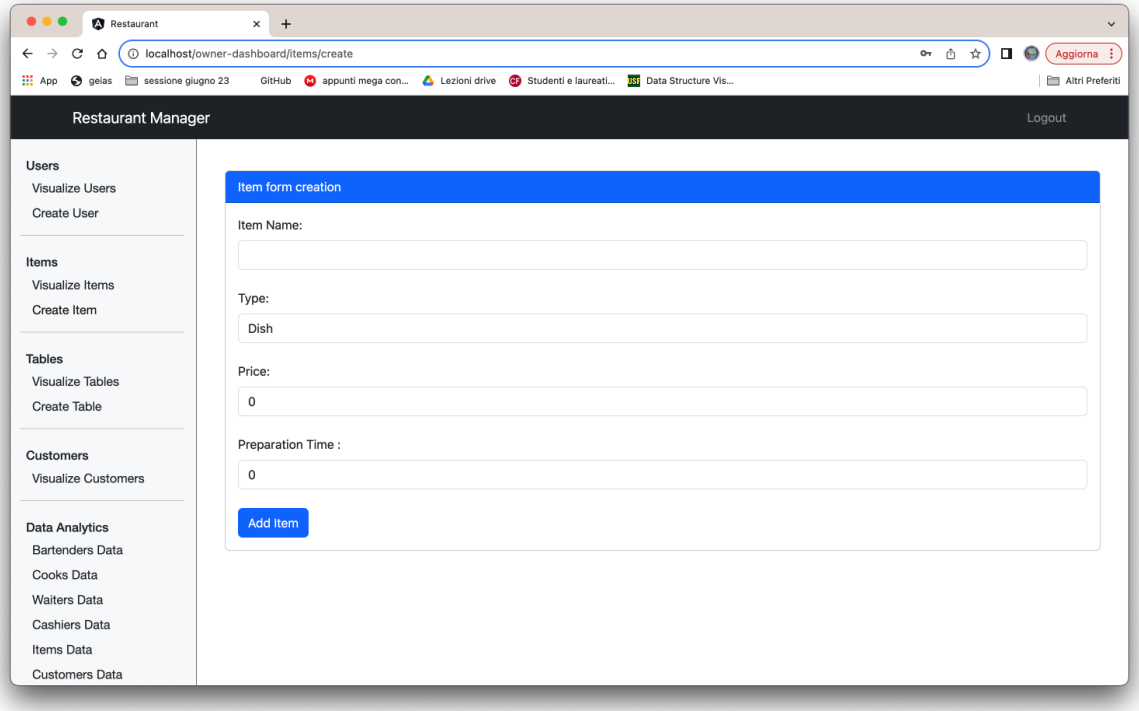
3.2.2 Create User



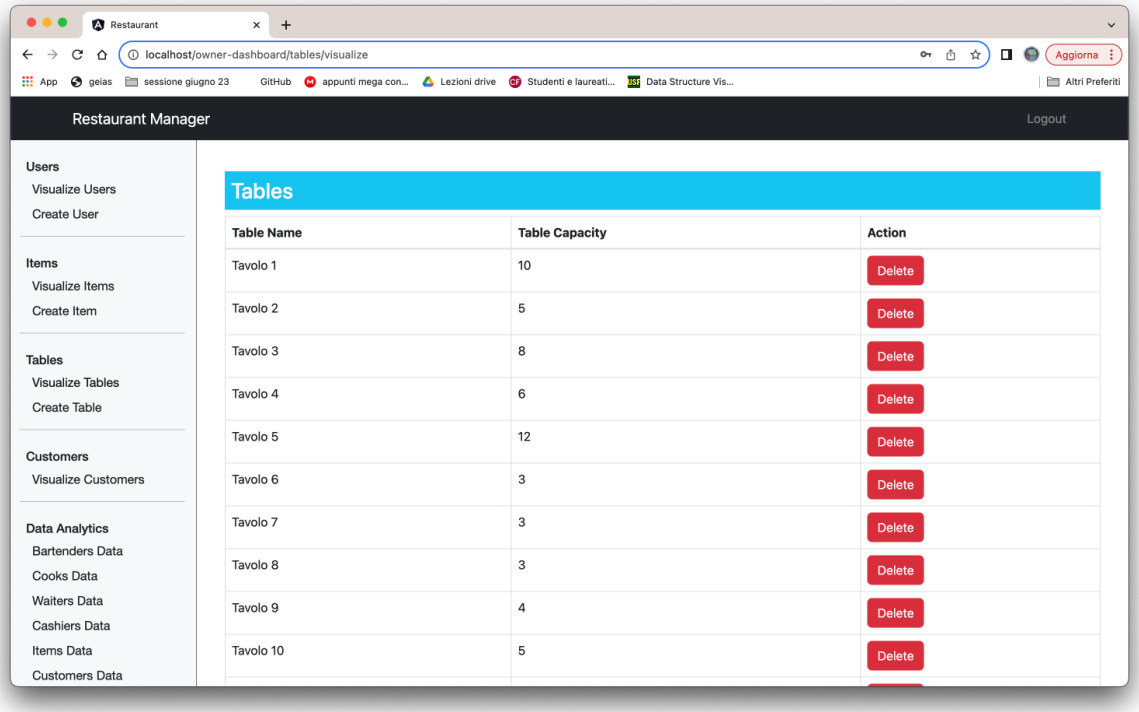
3.2.3 Visualize Items



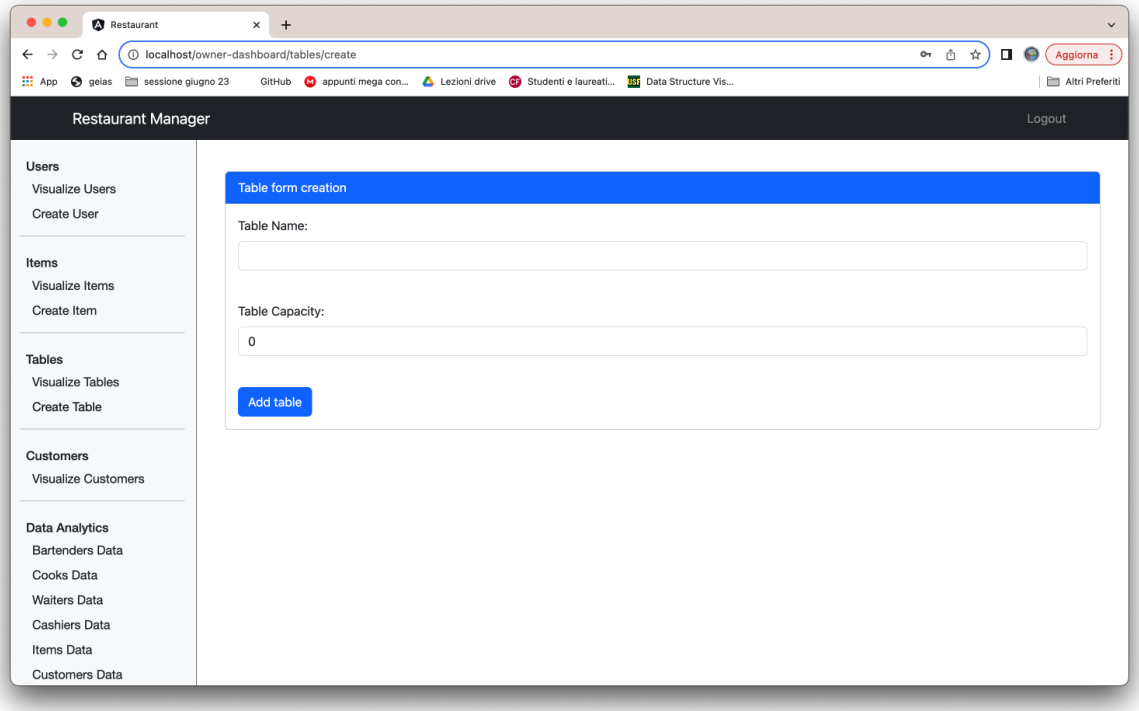
3.2.4 Create Item



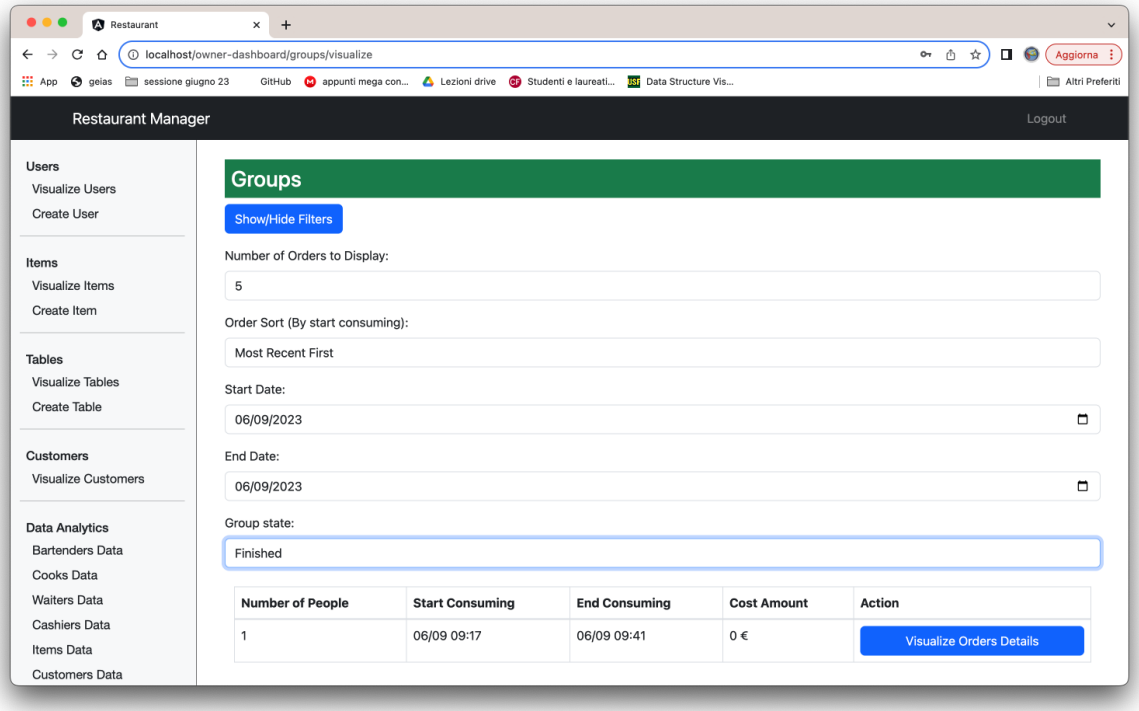
3.2.5 Visualize Tables



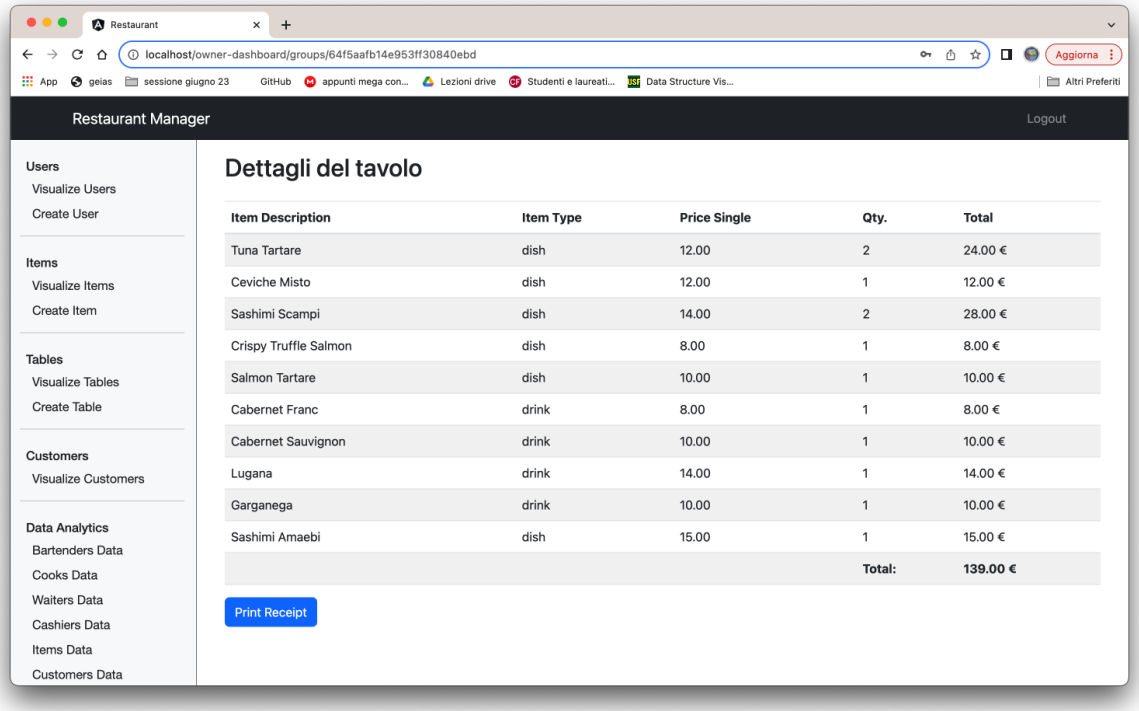
3.2.6 Create Table



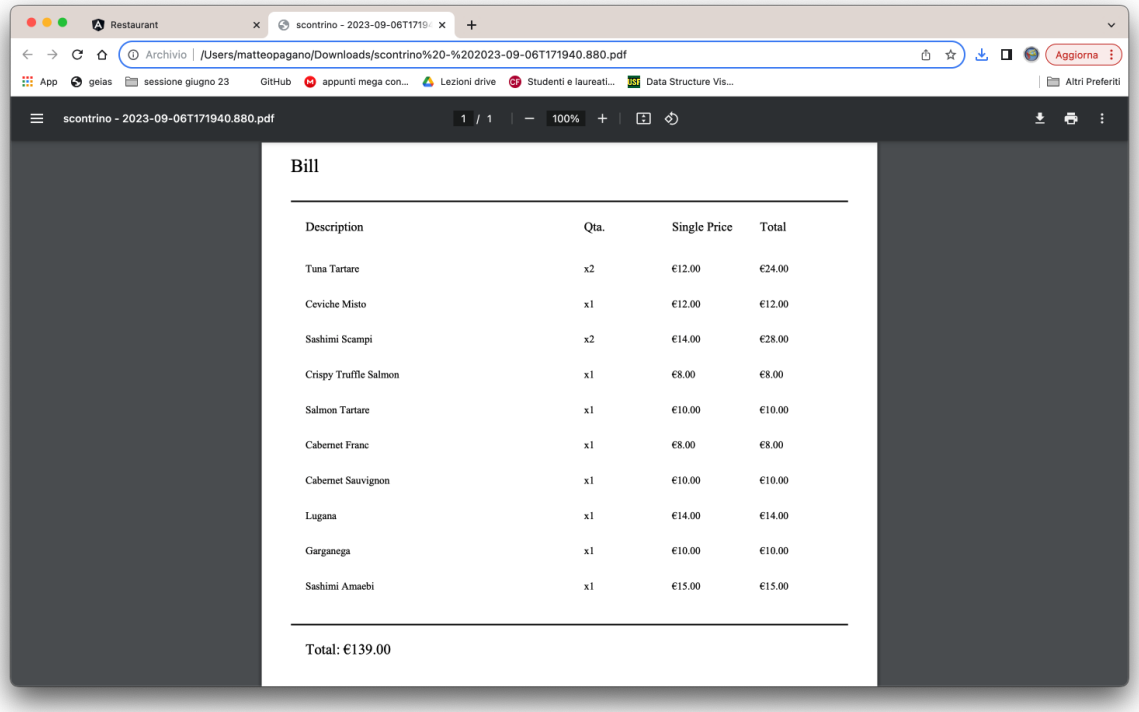
3.2.7 Visualize Customers



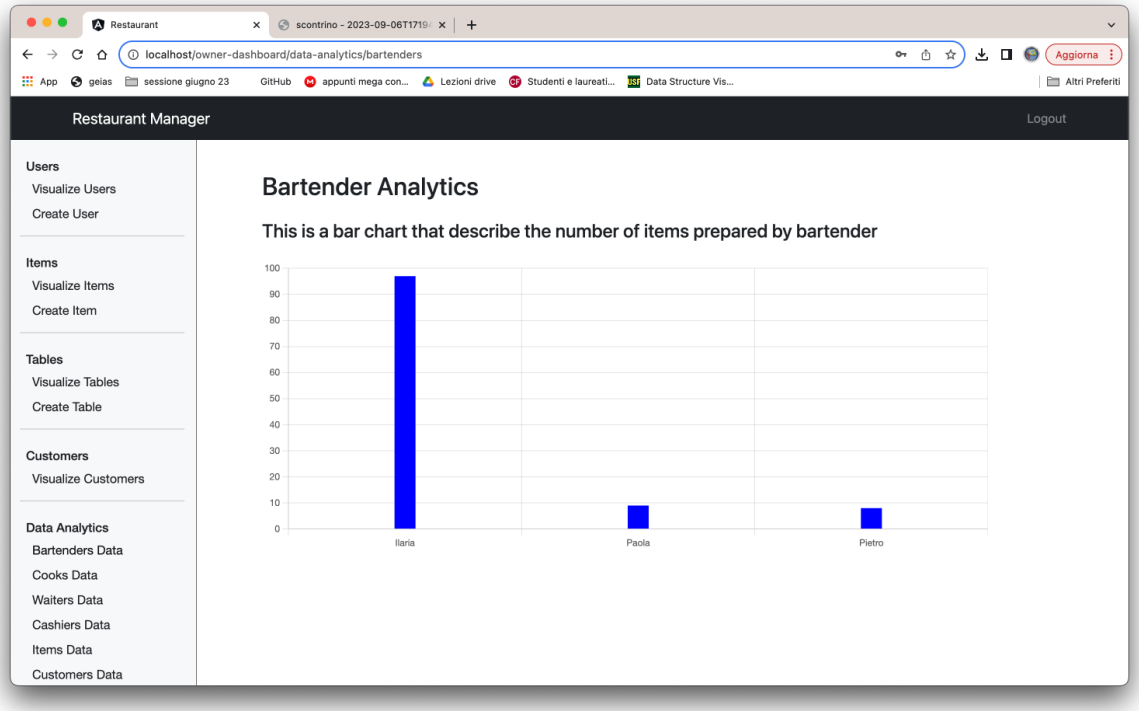
3.2.8 Visualize Customer orders



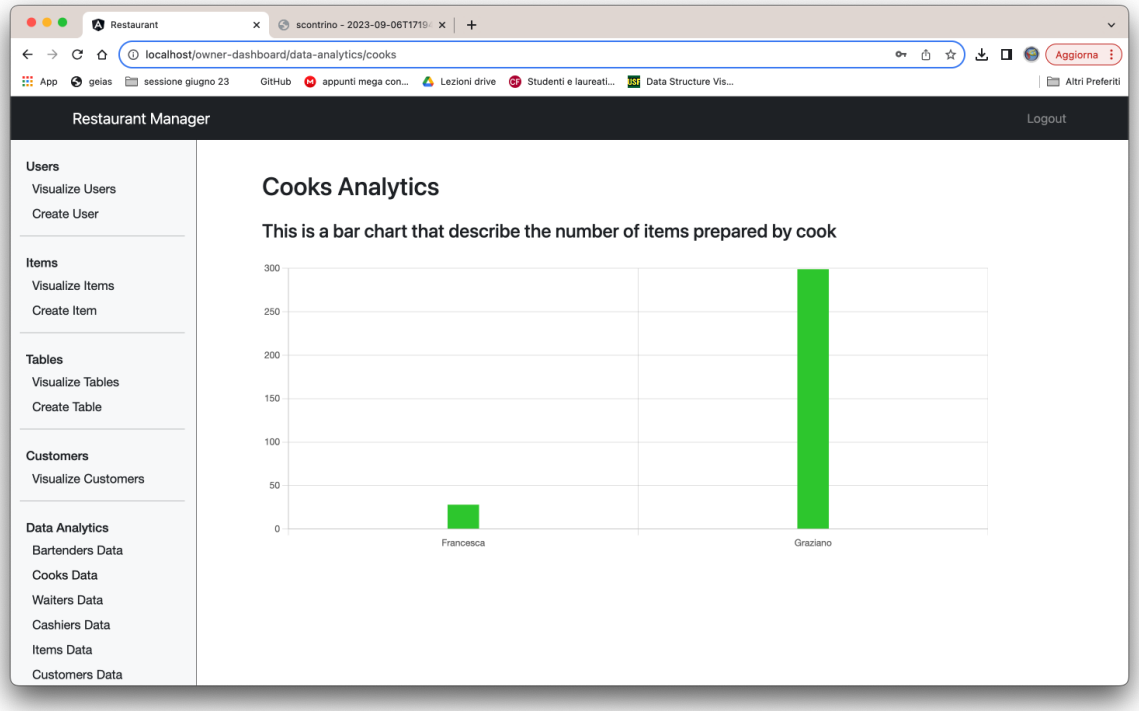
3.2.9 Visualize bill



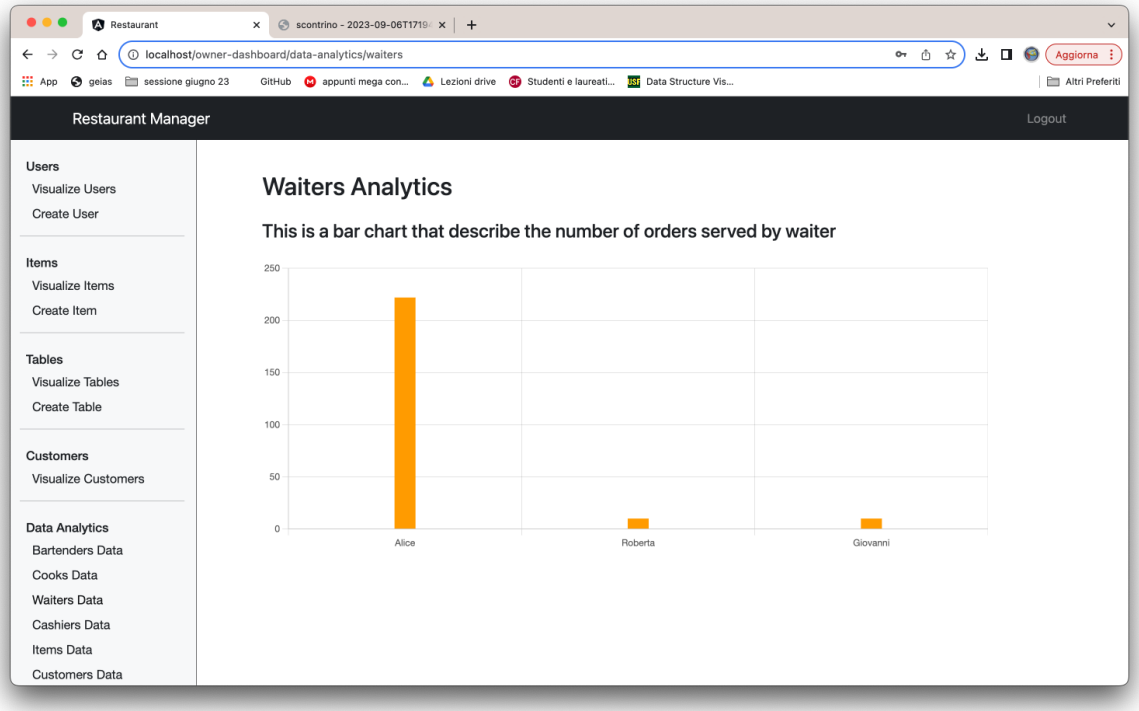
3.2.10 Visualize Bartenders data Analitycs



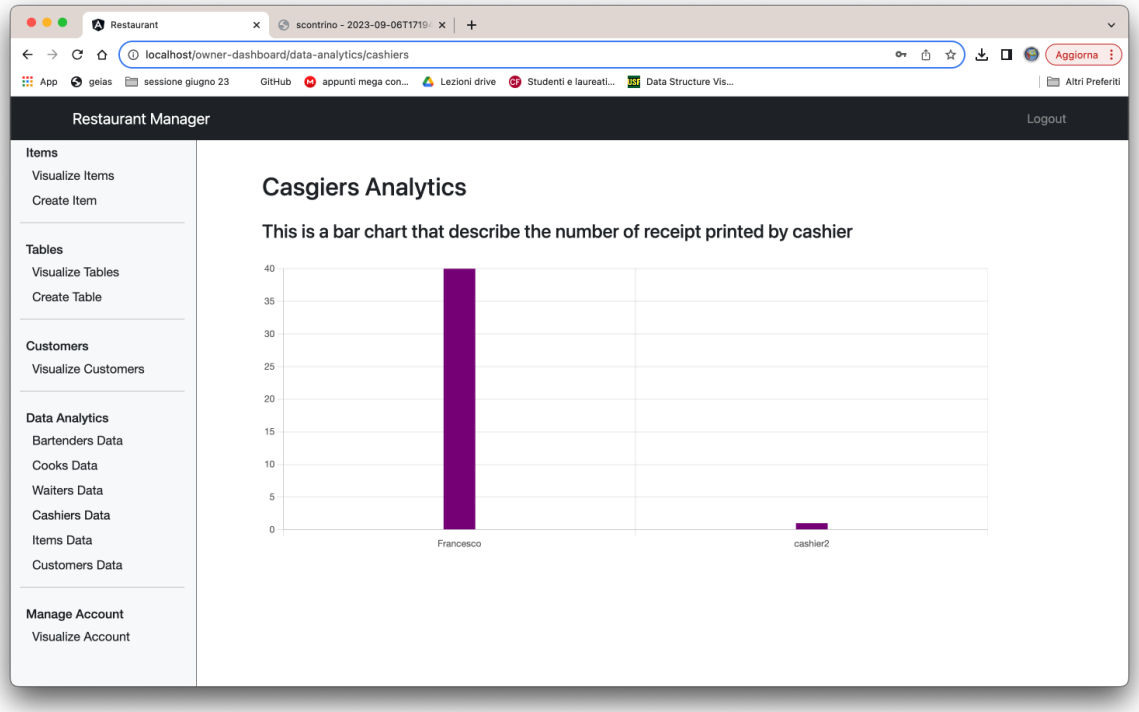
3.2.11 Visualize Cooks data Analitycs



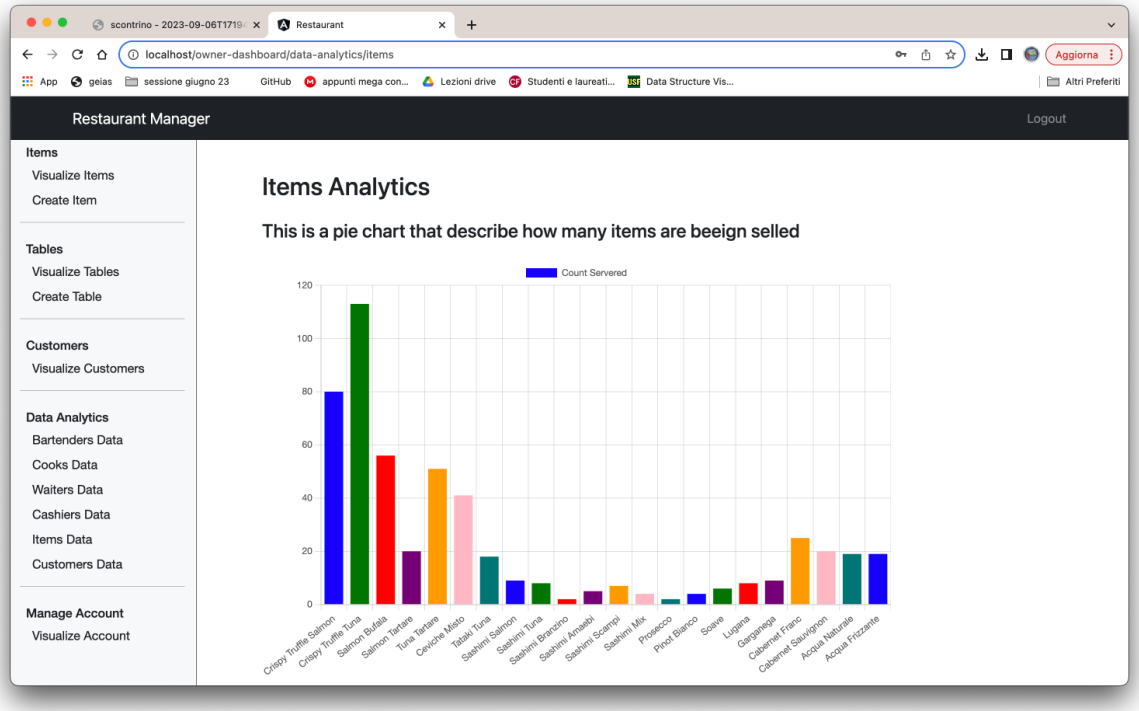
3.2.12 Visualize Waiters data Analitycs



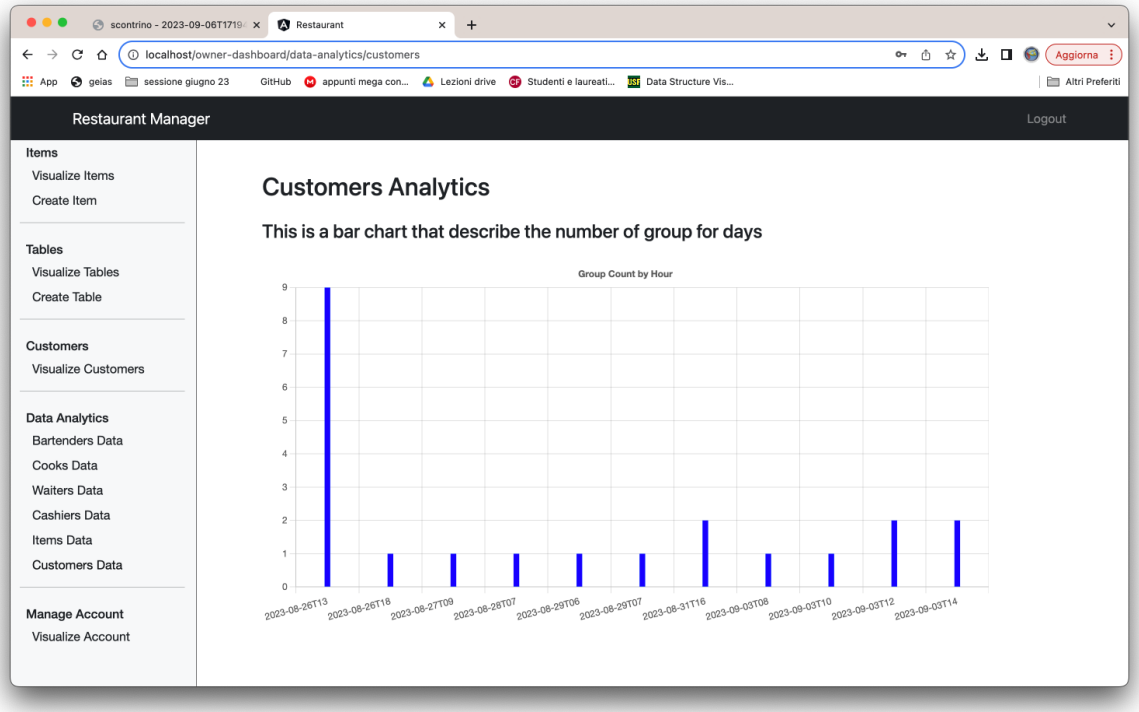
3.2.13 Visualize Cashiers data Analitycs



3.2.14 Visualize Items data Analitycs

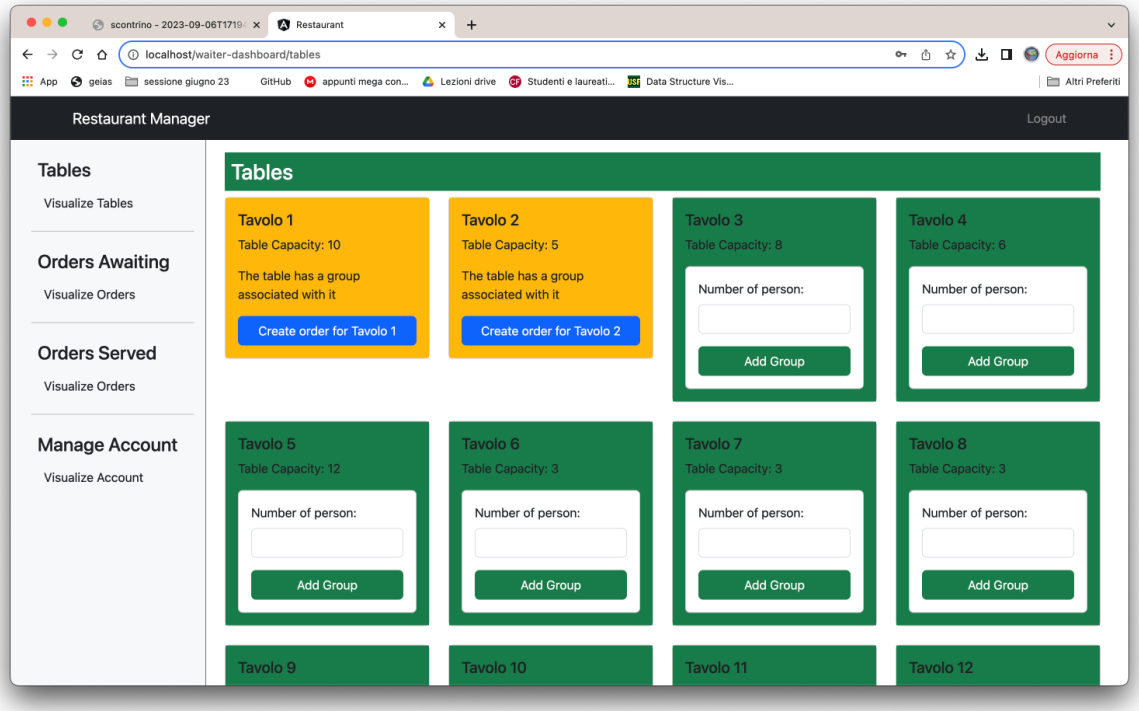


3.2.15 Visualize Customers data Analitycs

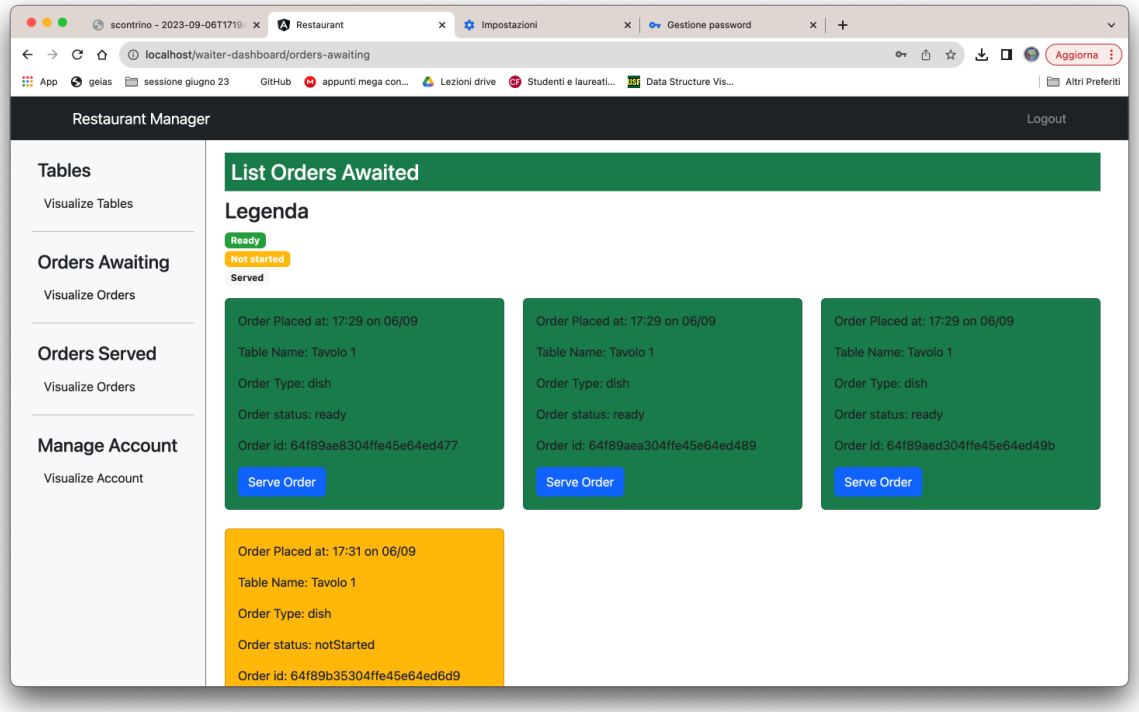


3.3 Waiter

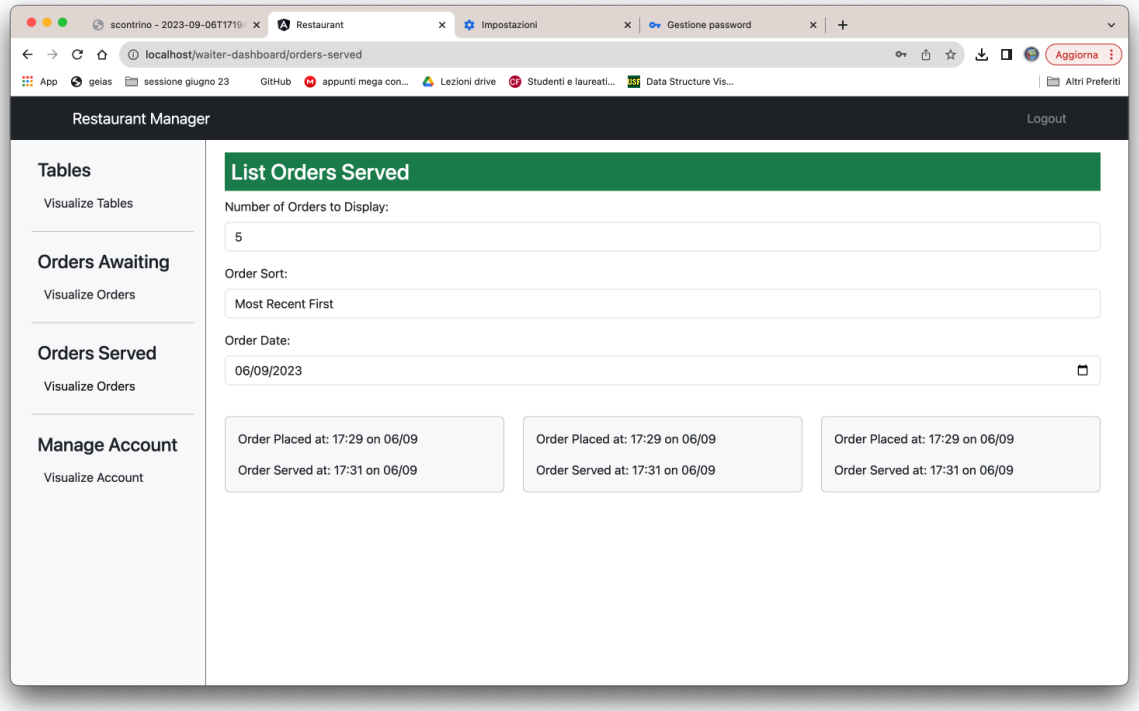
3.3.1 Visualize Tables



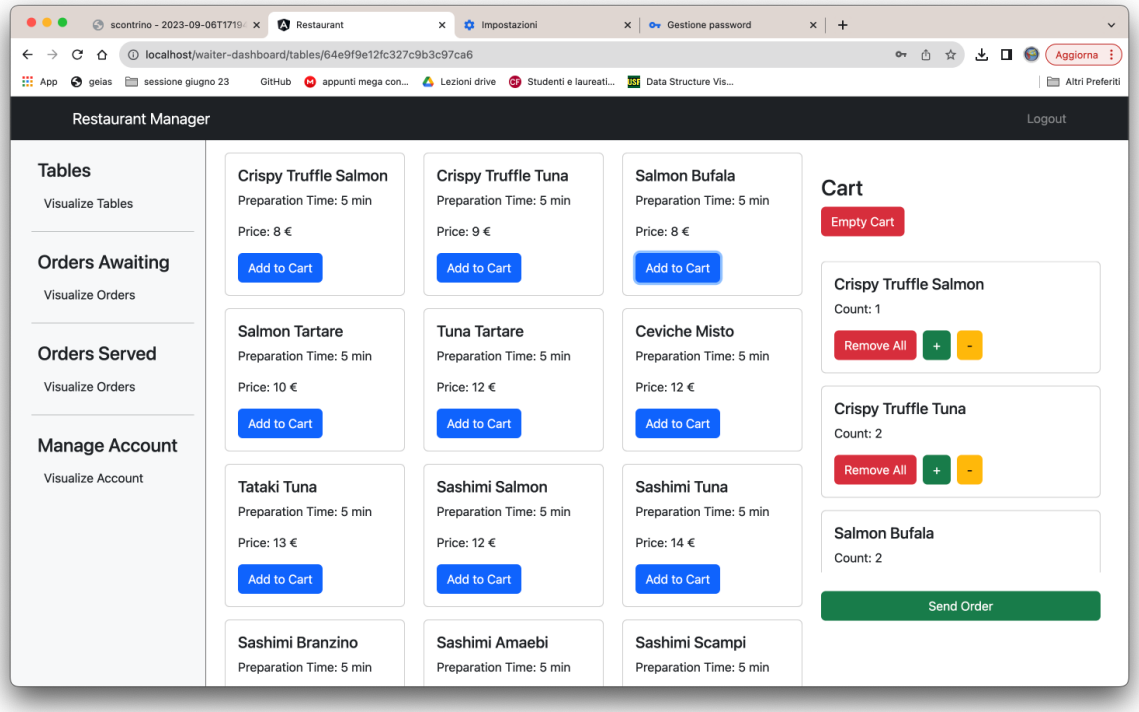
3.3.2 Visualize Orders Awaiting



3.3.3 Visualize Orders served

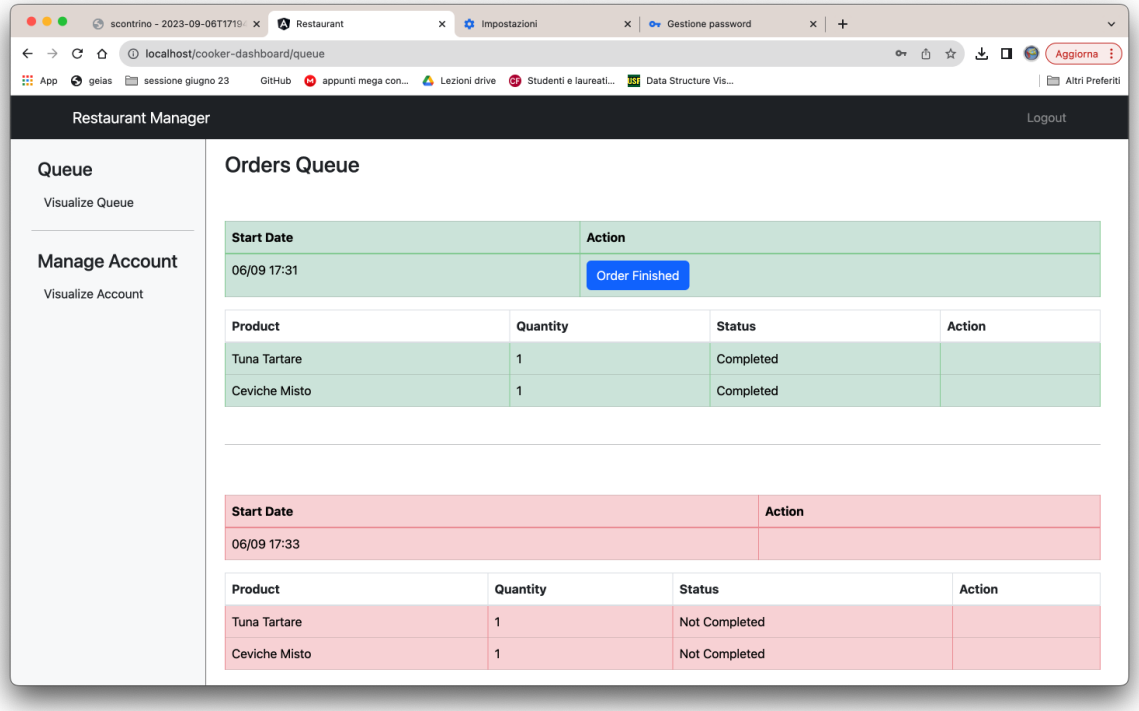


3.3.4 Create Order



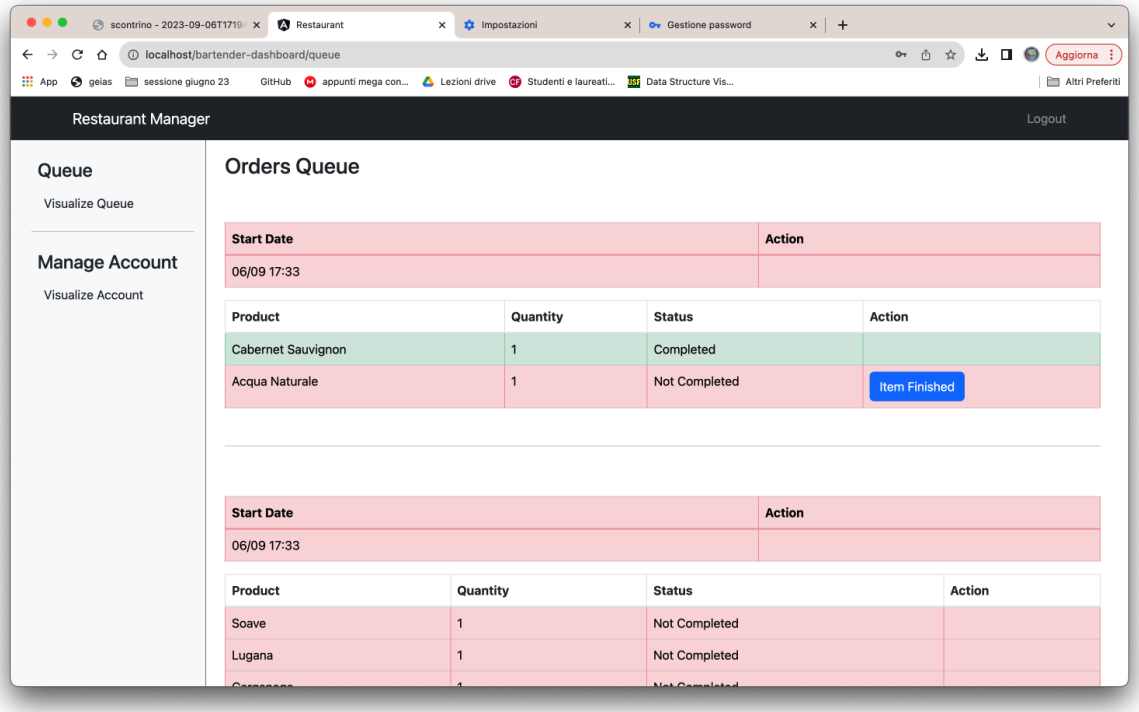
3.4 Cook

3.4.1 Orders Queue



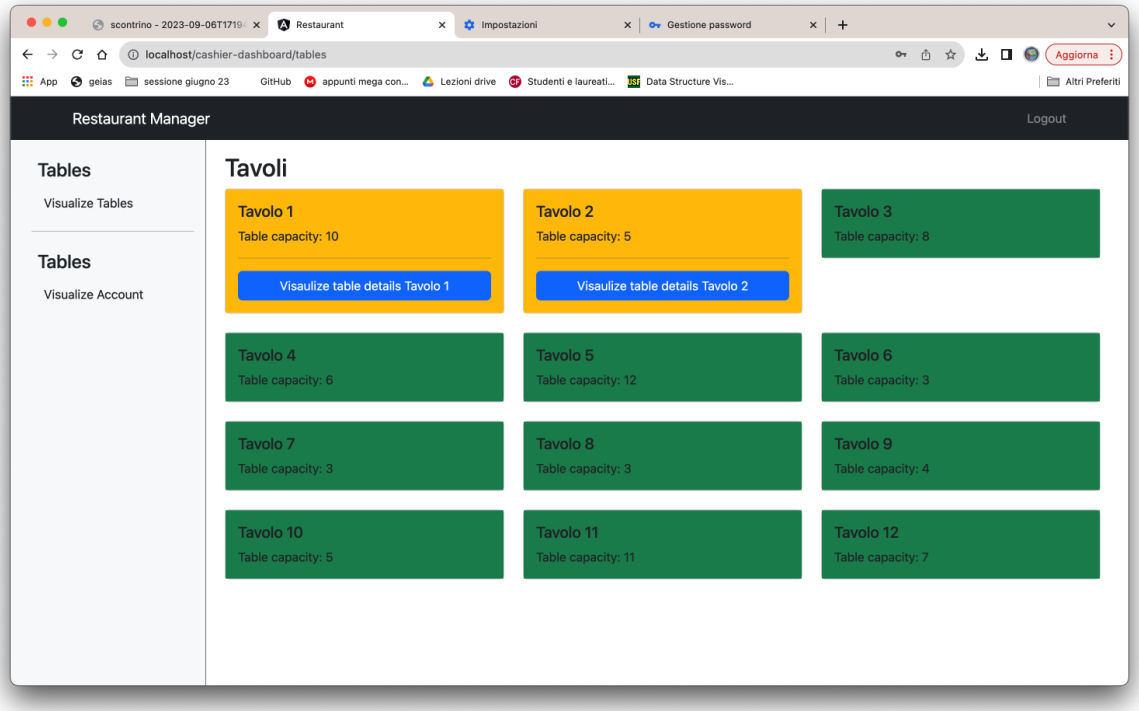
3.5 Bartender

3.5.1Orders Queue

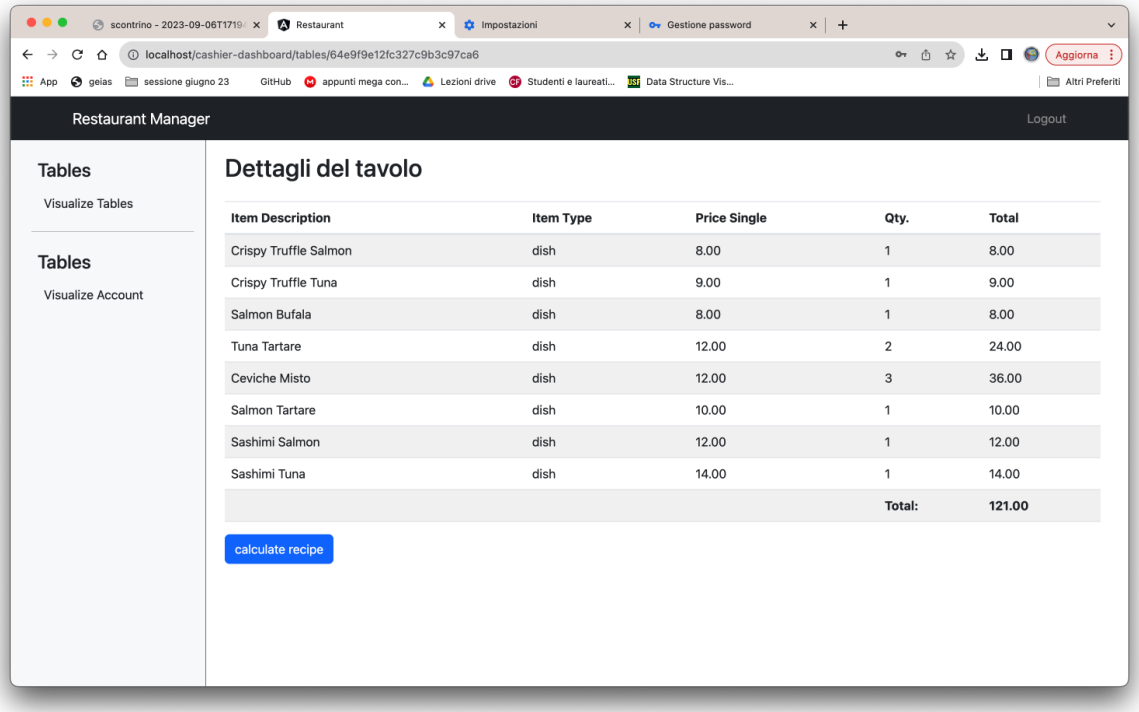


3.6 Cashier

3.6.1 Visualize Tables



3.6.2 Visualize Tables Details



3.7 Common

3.7.1 Manage Account